AFRL-IF-RS-TR-2005-274
Final Technical Report
July 2005

BUILDING SECURE AND RELIABLE NETWORKS THROUGH ROBUST RESOURCE SCHEDULING

Princeton University

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. J904

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2005-274 has been reviewed and is approved for publication

APPROVED: /s/
ALAN J. AKINS
Project Engineer

FOR THE DIRECTOR: /s/
WARREN H. DEBANY, JR.
Technical Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	July 2005	**Final** **May 00 – Aug 04**

4. TITLE AND SUBTITLE

BUILDING SECURE AND RELIABLE NETWORKS THROUGH ROBUST RESOURCE SCHEDULING

5. FUNDING NUMBERS

G - F30602-00-2-0561
PE - 62301E
PR - J904
TA - 01
WU - A1

6. AUTHOR(S)

Larry Peterson, Randy Wang, and Vivek Pai

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Princeton University
Office of Research and Project Administration
New South Bldg., P O Box 35
Princeton NJ 08544

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington VA 22203-1714

AFRL/IFGA
525 Brooks Road
Rome NY 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

AFRL-IF-RS-TR-2005-274

11. SUPPLEMENTARY NOTES

DARPA Project Manager: Col Tim Gibson/ATO/(703) 526-4764 tgibson@darpa.mil
AFRL Project Engineer: Alan Akins/IFGA/(315) 330-1869 Alan.Akins@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***

Providing fault tolerance and combating denial of service (DoS) attacks have traditionally been the research subjects of the fault tolerant computing community and the security community. This report takes a different perspective, one that applies a unified set of mechanisms and algorithms to the problem of protecting a network system from both failures and DoS attacks. The problem is viewed as a matter of resource allocation and management. Protection against DoS attacks is addressed as a special case of careful scheduling of time and fault tolerance is addressed as a special case of careful scheduling of space. A set of general scheduling mechanisms has been developed for both time and space. Specifically, the focus was on three key aspects of a networked system: (1) the local resources on each network router, (2) the network-wide resources applied to a given information service, and (3) the network bandwidth consumed by end-to-end flows.

14. SUBJECT TERMS

Fault tolerant networks, denial of service, resource allocation and scheduling, load balancing, computer networks, cyber defense

15. NUMBER OF PAGES 68

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

1 Introduction

The central thesis of this work is that robust network services that are immune to denial of service attacks (DoS) or faults must possess two important qualities: we call them *completeness* and *generality*.

There are two dimensions to completeness: one dimension is along the different types of resources on a *local* node. For example, in addition to protecting link bandwidth from DoS attacks, we must also protect resources such as routing tables and processing cycles in a router. Lack of adequate protection for any of these resources leaves an entire system vulnerable to attackers. The second dimension is along the different *distributed* components that must function properly in order to provide the desired level of service. These components may include, for example, network routers, name servers, and servers that implement the service. A successful attack against or failure of any of these components leaves the service beyond the users' reach.

The second important quality of robust network services is generality. Solutions that treat DoS attacks or faults as extraordinary events that require special case handling are not general. By grafting onto existing systems these extra sets of bells and whistles that are not needed during "normal" operations, we rarely get to exercise them and test their effectiveness. As a result, we have little confidence in their robustness when these obscure code paths do get triggered.

Providing fault tolerance and combating DoS attacks have traditionally been the research subjects of two distinct communities: the fault tolerant computing community and security community. This proposal takes a different perspective, one that applies a unified set of mechanisms and algorithms to the problem of protecting a networked system from both failures and DoS attacks. We view the problem as a matter of resource allocation and management. In our view, protection against DoS attacks is a special case of careful scheduling of *time*, and fault tolerance is a special case of careful scheduling of *space*.

When scheduling competing requests for the same resource at different times, we have three progressively more aggressive goals:

- *Protection* against DoS attacks–no single entity is allowed to monopolize a resource all the time;
- *Fairness*–each user gets a share of the resource based on some policy, such as one based on how much she has paid for it; and
- *Optimality*–scheduling is done in such a way that it optimizes some utility function, such as aggregate network utilization or profits earned by the service provider

.

Each of the latter goals subsumes the earlier ones. A "fair" system is necessarily "protected" because attackers are by definition committing an "unfair" act. An "optimal" system can easily support "fairness" and "protection" by defining its utility function appropriately. On the other hand, a "protected" system is not necessarily "fair", and a "fair" system is not necessarily "optimal".

Similarly, when choosing among several alternatives to satisfy a single request, we have three progressively more aggressive goals:

- *Fault tolerance*–there is enough redundancy in the system so that an equivalent alternative always exists when failure occurs;
- *Load balancing*–we choose an alternative in such a way so that all alternative systems are roughly equally loaded; and
- *Optimality*–we choose an alternative in such a way that optimizes some arbitrarily defined utility function, such as response time or throughput.

Each of the latter goals subsumes the earlier ones. A "load-balanced" system should be able to provide "fault tolerance" because faults are an extreme form of load imbalance. An "optimal" system can easily support "load balance" and "fault tolerance" by defining its utility function appropriately. On the other hand, a "fault tolerant" system is not necessarily "load balanced", and a "load balanced" system is not necessarily "optimal".

Having recognized the relationships among these goals, instead of designing a set of special bells and whistles to achieve the more limited goals, we developed a set of general scheduling mechanisms for *both time and space* so that we can achieve the more aggressive goals. Specifically, we focused on three key aspects of a networked system: (1) the local resources on each network router, (2) the network-wide resources applied to a given information service, and (3) the network bandwidth consumed by end-to-end flows. The following three sections review our results in these three areas.

2 Local Router Resources

The first major thrust of our work is to design a router architecture that both allows us to extend the set of functions beyond the traditional forwarding service, but do so without sacrificing robust performance. This allows, for example, routers to be programmed to filter packets, translate addresses, make level-*n* routing decisions, broker quality of service (QoS) reservations, thin data streams, run proxies, support computationally-weak home electronic devices, serve as the front-end to scalable clusters, and support application-specific virtual networks. In general, we expect routers to support a wide assortment of *forwarding functions*, each of which processes and forwards packets in a flow-specific way.

At the same time routers are being programmed to implement new services, emerging hardware is making it possible to build routers from commercial off-the-shelf (COTS) components, including system area network (SAN) switches [23], network processors [24, 54], and programmable line cards [2, 46]. In general, we expect non-core routers to be constructed from a rich topology of these components, coupled with general-purpose commodity processors.

The problem such a hardware landscape creates is one of mapping the packet flows that traverse a router—and by implication the forwarding functions that implement services they require—onto a particular hardware configuration. The standard systems response to this situation is to define a virtual router

architecture. This section describes such an architecture, called VERA, that hides the hardware details from the forwarding functions. VERA is designed with the following properties in mind:

Extensible: Because we are implementing *extensible* routers, our design must export an interface and protocol that allows new functionality to be easily added to the router.

Compliant: It can be tempting to overlook some of the requirements of RFC1812 [3] when creating a new router architecture. Our goal is to develop an architecture that supports all of the requirements of a compliant router, with special concern for broadcast and multicast.

Efficient: Subject to the extensibility and compliancy requirements listed above, we want the architecture to support efficient implementations on the given hardware. For example, by taking advantage of the processor on intelligent network interface cards (NICs), many packets can be completely processed and forwarded without involving the main processor at all. By offloading the main processor as much as possible, we leave extra headroom for user-level extensions.

In the development of VERA, we have made a series of design choices that, when taken together, provide a consistent and coherent framework. Nearly every design choice represents a trade-off among performance, complexity, and modularity. Because VERA is designed for implementing extensible internet protocol (IP) routers on a heterogenous processing environment based on COTS hardware, we have made significantly different design choices than either a general purpose operating system or a router operating system for a single, centralized processor. The main contribution of this effort is to identify and motivate these design choices.

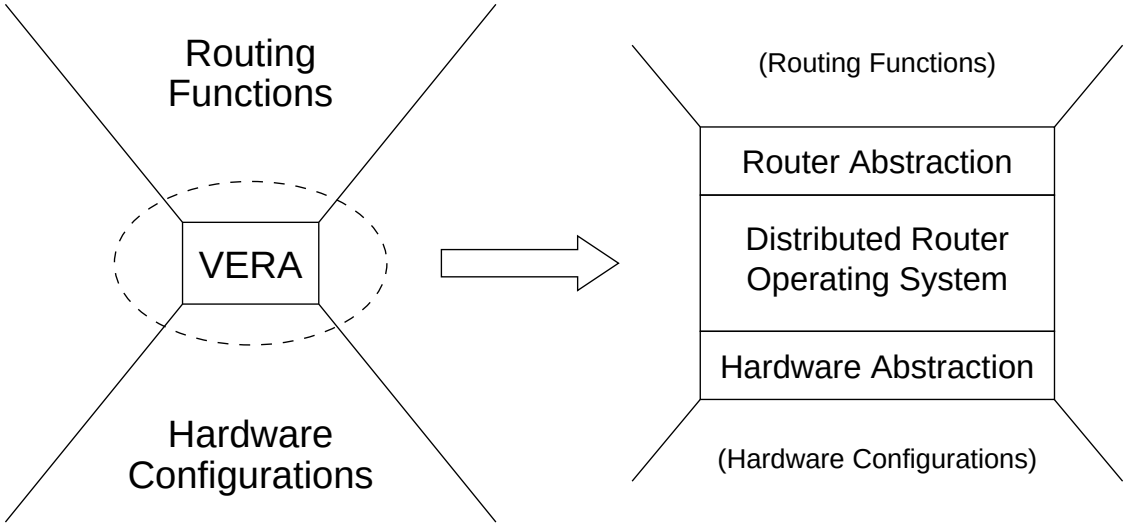


Figure 1: VERA constrains the space of routing function implementations and hardware exposure to facilitate the mapping between the two.

Figure 1 shows how the VERA framework constrains and abstracts the essence of both the routing function space and the hardware configuration space. VERA consists of a router abstraction, a hardware abstraction, and a distributed router operating system. The router abstraction must be rich enough to support the RFC1812 requirements as well as the extensions of interest. The hardware abstraction must be

rich enough to support the range of hardware of interest. However, it should expose only enough of the hardware details needed to allow for efficient router implementations. Note that both abstractions must be well "matched" to one another so that the map between them (i.e., the distributed router operating system implementation) is efficient and clean. The abstractions must also be chosen to allow us to model and reason about the system with adequate fidelity without also getting bogged down by details.

This section describes each of the three main components: the router abstraction (Section 2.1), the hardware abstraction (Section 2.2), and the distributed router operating system (Section 2.3). For each layer, we both describe the architecture of the layer, and report our preliminary experiences implementing the layer. Section 2.4 describes the **vera.o** device driver as it is implemented in Linux.

2.1 Router Abstraction

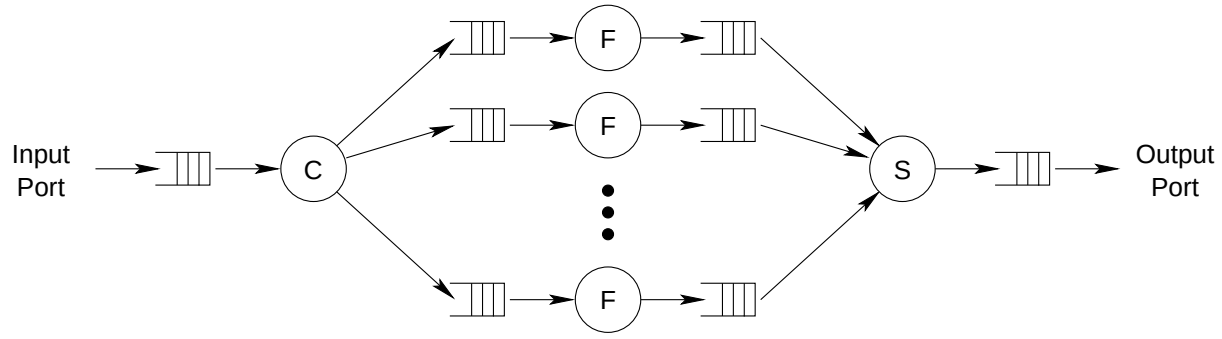


Figure 2: The classifying, forwarding, and scheduling of IP packets.

We define the router architecture visible to programmers writing new forwarding functions. The main attribute of the architecture is that it provides explicit support for adding new services to the router. Figure 2 shows the flow of packets in a router from an input port to an output port. A switching *path* is the instantiation of a forwarder, along with its associated input and output queues. An IP router performs three primary functions on each packet:

Classify: The *classifier* gets packets from an input port, creates an *internal routing header* for the packet, and (based on the contents of the packet) sends the internal routing header to the appropriate path(s). Classifiers map network flows onto switching paths. While classifiers do not modify packets, they can pass information to the path using fields in the internal routing header. The internal routing header is discussed further in Section 2.3.3. To support multicast and broadcast, classifiers can "clone" packets and send them along multiple paths.

Forward: The *forwarder* gets packets from its single input queue, applies a *forwarding function* to the packet, and sends the modified packet to its single output queue. All transformations of packets in the router occur in forwarders.

Schedule: The *output scheduler* selects one of its non-empty input queues, removes an internal routing header, and sends the associated packet to the output port. The scheduler performs no processing (including link-layer) on the packet.

The router abstraction hides details about the underlying hardware. The abstraction's model of the hardware is that of a single processor, a single memory pool, and no explicit switching elements. Details of data movement are hidden and all connections appear to be point-to-point. Classifiers, forwarders, and output schedulers run as separate threads. Note that output schedulers are different than the *thread schedulers*. The reason why we require that classifiers and schedulers not modify packets and why we restrict forwarders to a single input queue and a single output queue is motivated by our thread scheduling scheme; we discuss this later in Section 2.3.2.

At router initialization time, each port has an associated classifier and scheduler. In addition, there is an initial set of pre-established switching paths. To support QoS flows and extensions, our architecture allows paths to be dynamically created and removed by other paths. Section 2.1.2 gives details about this application programmer interface (API).

2.1.1 Classification Hierarchy

Our router architecture recognizes that packet classification is not a one-step operation. Instead, we view classification occurring in distinct stages. A simple classification sequence might be:

Sanity Check: the first step of classification is to identify packets which must be ignored or are malformed. Packets that are not identified as malformed are sent to the next level.

Route Cache: at this level, the packet is quickly compared against a cache of known flows to determine the correct path within the router. Packets not in the cache, as well as packets that require special processing (e.g., initial packets of a flow), are sent to the next level.

Prefix Match: most routers run a prefix matching algorithm that maps packets based on some number of bits in the IP header, ranging from just the destination IP address to the source/destination addresses and ports [16, 32, 50, 55].

Full Classification: eventually, packets which have not been classified in early stages will reach a "mop-up" stage which handles all remaining cases, including application-level routing. This stage is often implemented with arbitrary C code.

Figure 3 shows that the internal structure of a classifier is really a hierarchy of subclassifiers. Once a packet leaves the classifier C, the packets are *fully* classified—a specific path is the target.

For our architecture, we impose the restriction that the outputs from a classifier are unique. Referring to Figure 3, this would mean that an arrow from C_1 could not point to the same path as an arrow from C_2, for example.

Although we believe that this hierarchical structure is a fundamental aspect to packet classification on an IP router [44], the exact form of the hierarchy is often dictated by the processor hierarchy onto which it will be mapped. We return to this issue in Section 2.3.1.

2.1.2 Router API

This section outlines some of the router layer API function calls.

p* = createPath*(C, C_parms, F, F_parms, S, S_parms)

This function creates a new path, p, by instantiating a forwarder, F, and attaching it to the existing

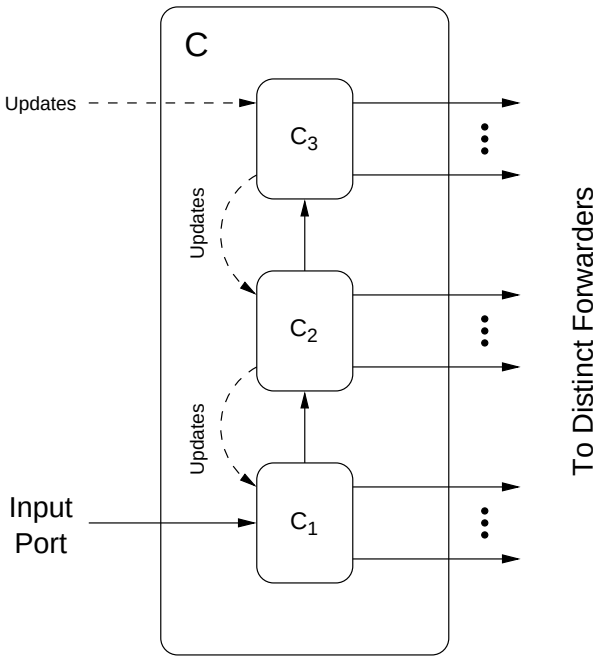


Figure 3: Classification Hierarchy. Classifier C is composed of partial classifiers C_1, C_2, and C_3. Solid lines indicate packet flow. Dashed lines indicate classification updates (e.g., route table updates).

classifier, *C*, and scheduler, *S* through new queues. (Note that *C* and *S* implicitly identify the input and output ports, respectively.) Figure 4 illustrates this process with an example. *C_parms* include the demultiplexing key needed to identify the flow which should be redirected to this path. *F_parms* are passed to the forwarder, and include the processor reservation (cycles, memory) required by the forwarder. *S_parms* include the link reservation needed by the output scheduler.

removePath*(p, parms)*

This function removes the existing path, *p*. The additional parameters indicate whether the path should be immediately terminated abandoning any packets in the queues or whether the path should be gracefully shut down by disconnecting it from the classifier first and then letting the packets drain out of the queues.

updateClassifier*(C, parms)*

This function allows updates (such as new routing tables) to be sent to a classifier.

2.1.3 Implementation

We have a complete implementation of the router architecture on a single processor with commodity NICs and have used it to build a variety of forwarders. The implementation is based on the Scout operating system [35]. We discuss OS-related issues in Section 2.3. At this stage, the interesting questions are (1) how Scout fills out the details not specified by the router architecture, and (2) whether or not the resulting system gave us the required flexibility in extending the router's functionality.

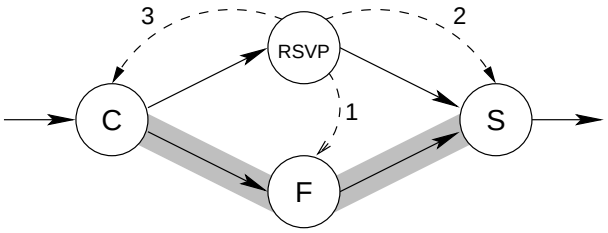


Figure 4: The basic operations performed by the **createPath** function. Here, an RSVP module (1) instantiates a new forwarder, then (2) attaches to the appropriate output scheduler, and finally (3) attaches to the appropriate classifier.

Regarding the first question, the most notable way in which Scout enhances the interface visible to the programmer is that it provides a component-based programming model for specifying forwarders. That is, Scout makes it possible to create a forwarder by combining IP modules with "extension" modules that modify standard IP forwarding. For example, we have constructed a forwarder that selectively drops packets during congestion for wavelet-encoded video [31]. This forwarder is specified as **eth/wavelet_drop/ip/eth**, where each of **eth**, **wavelet_drop**, and **ip** are names of modules that can be composed. In addition to defining the function that is applied to each packet, Scout also allows the programmer to specify how many CPU cycles and how much link bandwidth is to be allocated to a forwarder.

As to the second question, we have implemented a variety of forwarding functions, including both "fine-grain" extensions to IP—analogous to Router Plugins [15] and Click modules [34], and typified by the wavelet dropper example mentioned above—and "coarse-grain" extensions such as TCP proxies (e.g., **eth/ip/tcp/http_proxy/tcp/ip/eth**). Our router abstraction also supports active networking environments. We encapsulate the ANTS execution environment [57] in a forwarder specified as **.../udp/anep/nodeos/ants/nodeos/udp/...**), where **anep** is an active networking protocol layer and **nodeos** defines a wrapper (an execution environment) that isolates untrusted active protocols downloaded into ANTS from the rest of the system [43]. Beyond a variety of forwarding functions, we have also successfully used the interface to establish both best effort and QoS packet flows.

One additional issue that arises from our experience has to do with managing the classification hierarchy. It is often the case that a flow can only be classified by an upper level of the hierarchy, meaning that we need to ensure that lower levels do not. For example, suppose C_i is designed to match layer-4 patterns and C_{i-1} holds a route cache. If a call to **createPath** attaches to the classifier and specifies a level-4 pattern, we need to update C_i to add the new layer-4 pattern and also ensure that there is a miss in the route cache at C_{i-1}. The right side of Figure 5 shows how a cache will obscure higher classification levels; any change to the tree will require that the cache be updated to remain consistent.

2.2 Hardware Abstraction

This section describes the hardware abstraction layer for VERA. The object of any hardware abstraction layer (HAL), is to define an interface between the hardware and the "device independent" upper level software (typically, an operating system). This allows us to support a variety of hardware configurations without rewriting the operating system. Choosing the appropriate level of abstraction is somewhat of an art. We want to choose the abstraction level so that everything below is part of a consistent and uniform HAL and everything above does not directly access the hardware. If we select too high a level of abstraction, each port to a

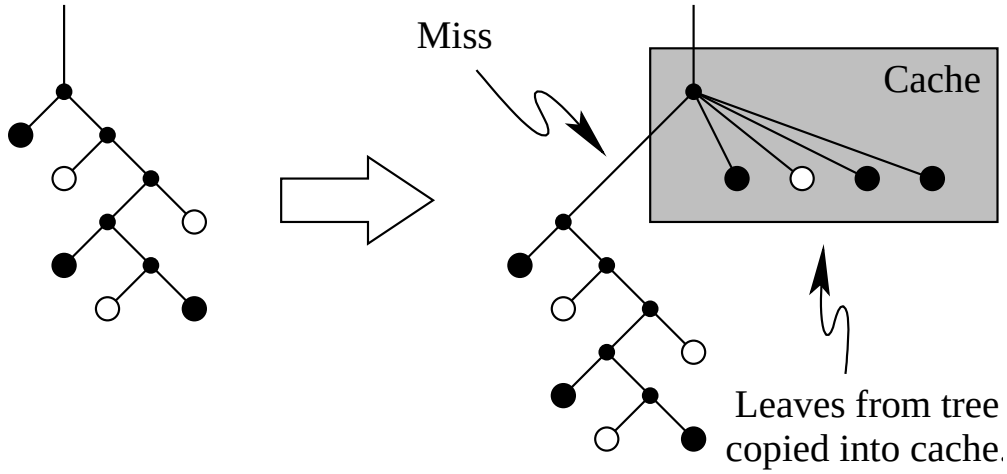


Figure 5: A partial classifier acting as a route cache. The left side shows a routing decision tree: internal nodes are decision points, leaves are path selections. The right side introduces a route cache. Misses must traverse the master tree.

new hardware configuration will require a major effort. If we select too low a level of abstraction, we will not be able to take advantage of higher-level capabilities provided directly by the hardware without breaking through the abstraction layer.

2.2.1 Example Hardware

Before discussing the architecture in detail, we present block diagrams for two example hardware configurations. This gives us a concrete reference when discussing the abstraction.

Figure 6 shows an example 4-port router based on commercially available components. (Details on the components are in Section 2.2.4.) While this router configuration is small, it has two especially interesting features: (1) a high processor cycle to port bandwidth ratio, and (2) substantial processing cycles "close" (low-latency) to the ports.

Figure 7 shows a larger configuration with thirty-two 100 Mbit/s Ethernet ports based on the IXP1200 network processor [24], the IX Bus, and the IXB3208 bus scaling fabric. By using four IXB3208 chips, this design scales to 128 ports and eight IXP1200 processors.

In our architecture, we also consider loosely coupled clusters of personal computers that utilize gigabit Ethernet, InfiniBand, or some other SAN technology as the underlying switching technology.

2.2.2 Hardware Abstraction

The hardware abstraction layer for VERA can be broken down into three major sections: processors, ports, and switches. The abstractions for the ports and processors are fairly standard. The abstraction for the switching elements is more involved and represents the bulk of the hardware abstraction layer. We describe each of the abstractions here:

Processors: The hardware abstraction layer groups the actual processors into virtual processors. Each virtual processor is either a single processor, a symmetric multiprocessor (SMP), or a complex/hybrid

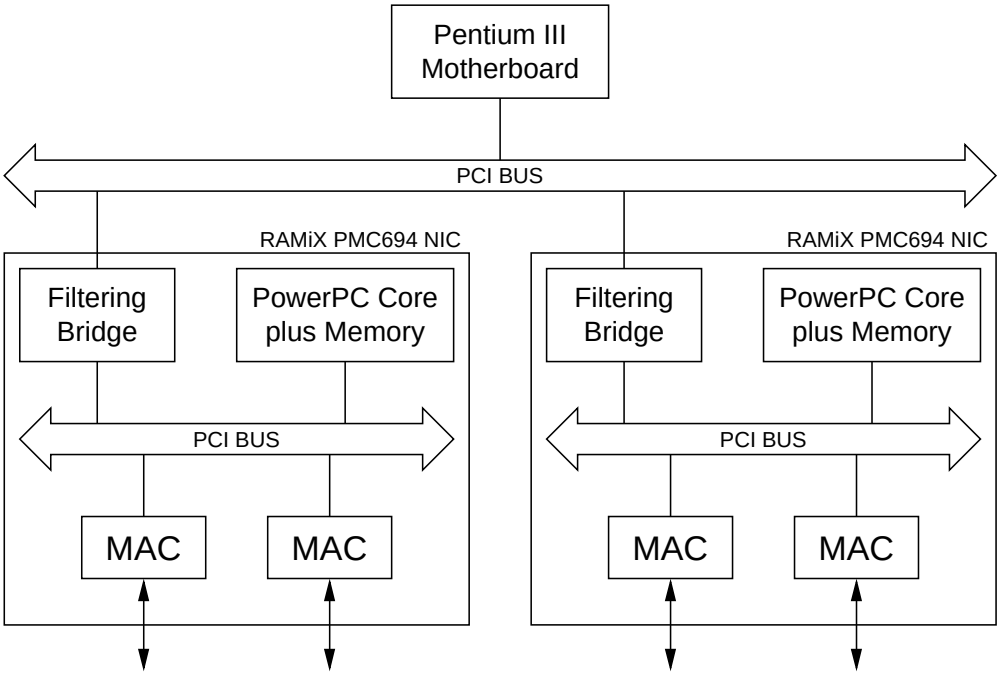


Figure 6: A four-port router using embedded PowerPC cores as network processors with a Pentium III as the master processor.

processor like the IXP1200 (which has a StrongARM core and six programmable microengines). The relevance is that each virtual processor is its own *scheduling domain* with a single thread pool. Also, any memory local to a processor is consolidated with and owned by that processor.

Ports: A device driver hides the register level details of the media access controller (MAC) chips and provides a uniform interface for upper layers. Each MAC chip is "owned" by a particular processor. The interface exports a scatter/gather capability that can read and write the header and data from separate memory locations. Note that these memory locations must be local to the processor that controls the port.

Switches: The switching elements are modeled as passive (no processing cycles) and shared (all devices connected to a switch share a single bandwidth pool). This also means that there is no explicit control registers accessible to the software to schedule data movement through the switch. VERA's switch abstraction provides an interface for interprocessor data movement and *distributed queues* whose head and tail are on different processors. Together, these are the primitives needed by the distributed router operating system to implement the interprocessor communication and message passing which is the basis for all the data movement within the router. Sections 2.2.4 and 2.3.4 discuss the details of data movement and queues in greater detail.

In addition to the abstractions of the processors, ports, and switches, the hardware abstraction maintains a static database containing the topology of the system, as well as the capabilities of the components.

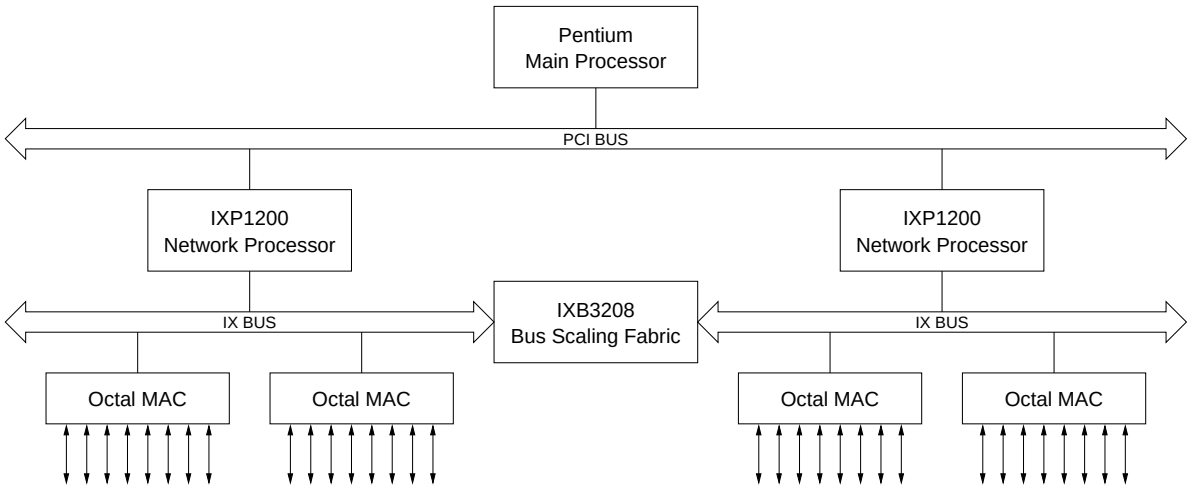


Figure 7: A hypothetical 32-port router using IXP1200 network processors with a Pentium as the master processor.

The router topology is the connected graph of the *direct* connections among the components of the router. By ignoring the switches and considering only the processors and ports, we can find a spanning tree with the *master processor* at the root and all the ports as leaves. This spanning tree is called the *processor hierarchy*. Figures 8 and 9 show the hardware abstraction of the architectures shown in Figures 6 and 7, respectively. The nodes of the graphs are the processors, switches, and ports (MACs). The graph edges (solid lines) indicate packet switching paths. The dashed arrows indicate the edges of the spanning tree defining the processor hierarchy. Figures 8 and 9 both have three distinct switching elements because both the filtering bridges of Figure 6 and the IXB3208 of Figure 7 segregate the buses and partition the bandwidth.

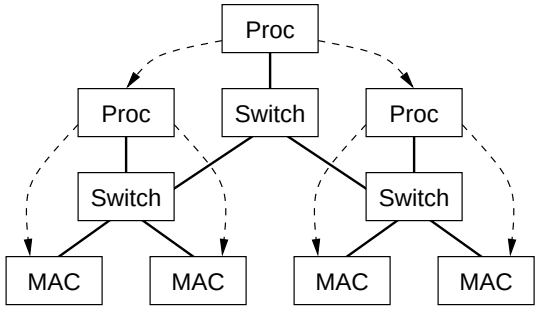


Figure 8: The hardware abstraction of Figure 6. The solid lines indicate packet flow paths. The dashed arrows show the processor hierarchy.

As mentioned above, the hardware abstraction layer maintains a static database of the component capabilities. The capabilities include the bandwidth of the ports and switches as well as the processing cycles available to the upper software layers. Specifically, this information is used by the distributed router operating system to determine appropriate placement of threads. It is important to note that the HAL only adver-

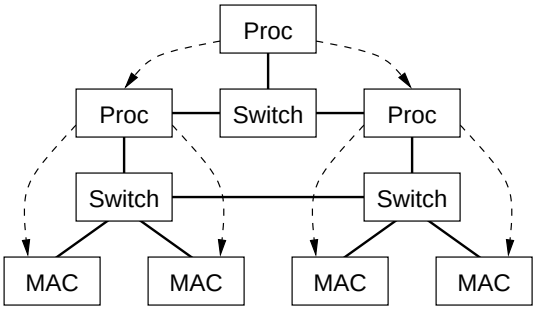


Figure 9: The hardware abstraction of Figure 7. The solid lines indicate packet flow paths. The dashed arrows show the processor hierarchy.

tises *available* processing cycles and switching capacity; this takes into account any cycles and switching capacity needed to implement the abstraction layer itself.

2.2.3 Hardware API

This section outlines some of the hardware layer API function calls. The following two functions hide details of how the hardware moves data:

putData*(local, remote, size)*
: This function pushes a block of data of length *size* from a *local* address to a *remote* address.

getData*(remote, local, size)*
: This function pulls a block of data of length *size* from a *local* address to a *remote* address.

The following three functions hide details of how the hardware queues data:

q* = allocHWqueue*(dir, depth)
: This function allocates a distributed queue, *q*, from a fixed-size pool of queues. The queue is configured to hold *depth* entries each of which is a pointer-sized integer. The direction parameter, *dir*, can be set to either **`incoming`** or **`outgoing`**. An incoming queue supports multiple remote writers and an outgoing queue supports multiple remote readers. Note that the local processor (which made the call to **allocHWqueue**) cannot write to incoming queues and cannot read from outgoing queues.

push*(queue, item)*
: This function pushes the *item* (a pointer-sized integer) on the given *queue*. The *queue* must be either a locally allocated outgoing queue or a remotely allocated incoming queue.

item* = pop*(queue)
: This function pops the *item* (a pointer-sized integer) from the given *queue*. The *queue* must be either a locally allocated incoming queue or a remotely allocated outgoing queue.

2.2.4 Implementation Issues

We now discuss two implementation issues—data movement and queue management across the PCI bus—for the PMC694 NIC and the IXP1200 Ethernet Evaluation Board (EEB). The key property of these configurations is that they employ a (PCI) bus-based switch. We later briefly comment on the implementation issues that arise on configurations that use other switching technology.

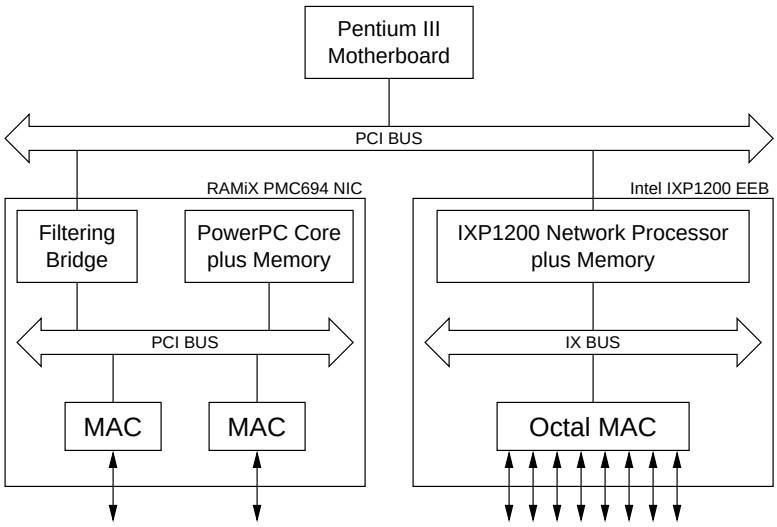


Figure 10: Testbed based on a Pentium III motherboard with both a PMC694 NIC and an IXP1200 EEB.

Figure 10 shows our development testbed consisting of a commodity motherboard connected to two different off-the-shelf network interface cards using a standard 33 MHz, 32 bit primary PCI bus. The motherboard is an Intel CA810E with a 133 MHz system bus, a 733 MHz Pentium III CPU, and 128 Mbytes main memory.

The first NIC is a RAMiX PMC694 [46]. It contains its own 266 MHz PowerPC processor, two 100 Mbit/s Ethernet ports, and 32 Mbytes of memory. The primary PCI bus (of the motherboard) is isolated from the secondary PCI bus (of the PMC694) with an Intel 21554 non-transparent PCI-to-PCI bridge. The secondary PCI bus is also 32 bits and operates at 33 MHz. The PMC694 has a two-channel direct memory access (DMA) engine and queue management hardware registers used to support the Intelligent I/O (I_2O) standard [26].

The second NIC is an Intel IXP1200 Ethernet Evaluation Board (EEB) which is sold as part of an evaluation kit [25]. The board contains a 199 MHz IXP1200 network processor, eight 100 Mbit/s Ethernet ports, and 32 Mbytes of memory. The IXP1200 chip contains a general-purpose StrongARM processor and six special-purpose MicroEngines. Like the PMC694, this boards also has a two-channel DMA engine and I_2O support registers.

Data Movement. The common denominator of all IP routers is that they move data from one network interface to another. The HAL exports an API that allows any of the processors in the router to either push (put) or pull (get) data to or from any of the other processors in the router.

Focusing on PCI-based configurations, we note that the PCI bus efficiency is highest when data is trans-

ferred in long *bursts*. This is a direct consequence of the fact that a PCI bus transaction consists of an *address phase* followed by one or more *data phases*. The address phase specifies the address of the transfer for the first data phase; each subsequent data phase within the same bus transaction occurs at the next higher address. The address phase takes one PCI clock cycle; in the best case, each data phase takes one PCI clock cycle. For each data phase, either the master or the target may insert *wait states* (each of which takes one PCI clock cycle). Note that read operations have a manditory wait state in their first data phase and each transaction must be separated by at least one PCI clock cycle. By transferring data in bursts (bus transactions with many data phases), the overhead (per byte transferred) of the address phase is reduced.

Because processors cannot always consistently generate these efficient bursts, they are often augmented with DMA engines specifically designed to use bursts to transfer blocks of data. Both the PMC694 and the IXP1200 have DMA engines which consistently generate burst transfers. Note that the Pentium motherboard does not have a high-speed PCI DMA engine.

Due to the freedom in the PCI specification, different board designs will exhibit different performance characteristics. We performed a series of measurements of the PCI bandwidth between the motherboard and each of the boards of Figure 10. Both the host processor (the Pentium III) and the on-board processor of the NICs can arbitrate for and then become the PCI bus master which allows them to initiate transfers. By using read or write instructions (or a DMA engine if available) a processor can pull or push data across the bus. Our experimental results are discussed in the following paragraphs.

Table 1 summarizes the results of our experiments with the PMC694. Here the Pentium processor runs a Linux 2.2 kernel while the PowerPC runs our "raw" code with no operating system. Under programmed I/O (PIO), we measured several different data movement techniques. These differ in the number of bits transferred per loop iteration (Transfer Size). We measured transfer size of 64 bits and 128 bits by unrolling the 32-bit loop two times and four times, respectively. We obtained our best times by using the ANSI C memcpy library routine (from the egcs-2.91.66 compiler) which uses tuned assembly routines. Our code was written in ANSI C.

Bus Master	Transfer Size	Mode	Direction	64-Byte Packets		1500-Byte Packets	
				Kpps	MByte/sec	Kpps	MByte/sec
host	32 bits	PIO	push	919.9	58.88	39.37	59.06
host	64 bits	PIO	push	975.0	62.40	41.58	62.37
host	128 bits	PIO	push	962.4	61.60	41.53	62.30
host	(memcpy)	PIO	push	1073.4	68.70	44.29	66.44
host	32 bits	PIO	pull	61.1	3.91	2.61	3.91
host	64 bits	PIO	pull	61.8	3.95	2.63	3.95
host	128 bits	PIO	pull	61.9	3.96	2.64	3.95
host	(memcpy)	PIO	pull	62.2	3.98	2.66	3.99
card	32 bits	PIO	push	365.5	23.39	14.84	22.26
card	64 bits	PIO	push	363.4	23.26	14.81	22.21
card	128 bits	PIO	push	363.9	23.29	14.81	22.21
card	—	DMA	push	534.2	34.19	17.57	26.35
card	32 bits	PIO	pull	53.7	3.44	2.29	3.43
card	64 bits	PIO	pull	53.8	3.44	2.28	3.41
card	128 bits	PIO	pull	53.8	3.44	2.28	3.41
card	—	DMA	pull	354.1	22.66	15.74	23.61

Table 1: Raw PCI Transfer Rates Between the PMC694 (card) and the Pentium III Motherboard (host).

Table 2 summarizes the results of our experiments with the IXP1200 EEB. These experiments are analogous to the PMC694 experiments. As with the PMC694 experiments, the Pentium processor runs a

Linux 2.2 kernel while the StrongARM runs our "raw" code with no operating system. We locally modified the firmware and jumper settings to run the board without an operating system and to allow it to be plugged into the PCI bus of a commodity motherboard. Note that the MicroEngines were not used in this experiment. (Details on our MicroEngine software architecture can be found in [48, 49].)

Bus Master	Transfer Size	Mode	Direction	64-Byte Packets		1500-Byte Packets	
				Kpps	MByte/sec	Kpps	MByte/sec
host	32 bits	PIO	push	1030.2	65.93	44.2	66.34
host	64 bits	PIO	push	1028.9	65.85	44.1	66.16
host	128 bits	PIO	push	1030.1	65.93	44.0	66.14
host	(memcpy)	PIO	push	1073.9	68.73	44.2	66.35
host	32 bits	PIO	pull	63.7	4.08	2.7	4.05
host	64 bits	PIO	pull	63.7	4.08	2.7	4.04
host	128 bits	PIO	pull	64.3	4.11	2.7	4.09
host	(memcpy)	PIO	pull	64.4	4.13	2.7	4.12
card	32 bits	PIO	push	349.2	22.35	14.7	22.12
card	64 bits	PIO	push	387.1	24.78	16.5	24.81
card	128 bits	PIO	push	387.1	24.78	16.5	24.81
card	—	DMA	push	204.7	13.10	32.7	49.01
card	32 bits	PIO	pull	69.1	4.43	2.9	4.42
card	64 bits	PIO	pull	69.8	4.47	3.0	4.46
card	128 bits	PIO	pull	71.1	4.55	3.0	4.53
card	—	DMA	pull	179.2	11.47	16.4	24.66

Table 2: Raw PCI Transfer Rates Between the IXP1200 EEB (card) and the Pentium III Motherboard (host).

From Tables 1 and 2, we see that the fastest way to move a packet from the card to the host and back to the card is for the card to push the packet to the host using its DMA controller and then have the host push the packet back to the card using memcpy-based PIO. (The notable exception is that due to the DMA setup overhead, small packets should be sent using programmed I/O on the IXP1200. Our experiments show that the crossover point is at approximately 168bytes/packet.)

Since there is only one DMA engine per NIC, this engine becomes a critical resource that is explicitly managed by the HAL. (An additional advantage of using DMA over programmed I/O is concurrency; after issuing a DMA request, a thread can either move on to other computations or yield to another thread.)

Moreover, because the PCI bus is a shared resource and there is no bandwidth reservation mechanism, the HAL must coordinate among the processors when using its DMA engine. For example, suppose processors A, B, C, and D are all connected to the same PCI bus. Suppose there is a QoS flow between processors A and B. In this case the HAL must ensure that a best effort flow between processors C and D does not prevent the QoS flow from meeting its reservation. To effectively support QoS flows in the router, the system must allow reservations on all of the limited, shared resources.

Thus, rather than simply hide the DMA engine beneath a procedural interface, we dedicate a server thread to the DMA engine on each processor. This thread coordinates with its peers to allocate the shared bandwidth available on the bus, as well as manages the local engine and DMA queues. The server supports two DMA work queues: a low-priority queue for packet traffic and a high-priority queue inter-processor control messages (e.g., messages that release packet data buffers.)

Because our hardware model allows multiple intervening processors and switching elements between any two processors, the intervening processors must store-and-forward the packets. While this appears to be the problem that the router as a whole is trying to solve (IP routing), the problem is significantly easier because, unlike the Internet, we have a centralized controller in the master processor. The HAL hides the

details of any necessary forwarding.

Queues. Another fundamental aspect of IP routers is that the packets of each network flow should be processed in FIFO order. Because we statistically multiplex the processing cycles among the ports, the router must occasionally buffer packets in distributed queues. Recall that a distributed queue has a tail located on one processor and the head located on a second processor.

Our requirement that the distributed queues support multiple readers and multiple writers on the remote side of the queue stems from the fact that we anticipate the number of available queues with hardware support to be limited. (In fact, the PMC694 only supports 2 queues in each direction at the interface to the PowerPC core.) Due to this limitation, we anticipate that these queues will need to be shared. We revisit this issue in Section 2.3.4.

Without special hardware support, implementing a multiple reader or writer queue over the PCI bus would require the processors to acquire and release semaphores using software techniques (for example, spinning on remote memory locations [21]). This is because the current version of the PCI bus specification [42] no longer supports bus locking by arbitrary bus masters. Fortunately, components that support I_2O have hardware registers that directly implement multiple reader or writer queues. They do this by hiding a FIFO queue behind a single PCI mapped register. When a processor reads the hardware register, there is a side effect of updating the queue pointer. The read and update happen atomically.

Because we take advantage of I_2O support as the basis for **push** and **pop**, we also must live with the restrictions of I_2O. Specifically, the local processor cannot access the registers used by the remote processors and vice versa. This is the reasoning behind the corresponding restrictions in **push** and **pop**. The benefit is that the implementation of **push** and **pop** can be as simple as a memory-mapped register write and read, respectively.

Other Switching Hardware. Our implementation effort up to this point has focused on the PCI bus as a switching element, but we considered other technologies when defining the HAL. There are two considerations when using other switches. First, the HAL defines operations for both pushing and pulling data across the switch. This is a natural match for a bus, which supports both read and write operations, and is consistent with interfaces like VIA [10] (which form the basis for future interfaces like InfiniBand [23]). On a switch component that supports only push (send) operations (e.g., an Ethernet switch), pull will have to be implemented by pushing a request to the data source, which then pushes back the data.

Second, the bus offers only best-effort, shared bandwidth. On a switching element that supports either dedicated point-to-point bandwidth (e.g., a crossbar) or virtual point-to-point bandwidth (e.g., InfiniBand channels), the role of the DMA server thread diminishes. In effect, the hardware supports the point-to-point behavior that the HAL requires. Of course a best effort switch (e.g., Ethernet switch) will have to be managed much like a PCI bus.

2.3 Distributed Router Operating System

As shown in Figure 1, the job of the distributed router operating system is to provide an execution environment for the forwarding functions which ties together the router abstraction and hardware abstraction. A case in point is that the OS must bridge the semantic gap between the high-level router abstraction and the low-level HAL; the OS *implements* the **createPath** call using the data movement and hardware queue support functions of the HAL. In addition to tying together these core abstractions, the OS provides a com-

putation abstraction in the form of a thread API and a memory abstraction in the form of both a buffer management API and an internal routing header datatype.

We use the Scout operating system as a prototype implementation of VERA for the special case of a single switch connecting a single processor to multiple MAC devices. In the following subsections, we outline the major features and abstractions provided by the operating system. We are in the process of extending Scout with VERA support for distributed configurations.

2.3.1 Processor Hierarchy

In our abstraction, the processors are organized into a hierarchy. The dashed arrows in Figures 8 and 9 shows how the master processors directly control the subordinate processors which eventually control the MAC devices. At router initialization time, administrative message queues are created from the master processor to each of its child processors, and so on down the hierarchy. The single, master processor maintains the master copies of the routing tables and controls the overall operation of the router. Since each processor operates independently and has its own thread scheduler, the control the master processor has over the other processors is, by necessity, coarse grained.

The processor hierarchy nicely maps to the classification hierarchy of Figure 3. The first partial classifier, C_1, always runs on the processor controlling the MAC for that input port. The last partial classifier, C_n, always runs on the master processor. Intermediate classification stages can be mapped to either the current processor, or the next processor up the processor hierarchy. Each classifier exports a single interface. This interface is only exported on the master processor. Updates to the classifier are made via this interface and then trickle down the classification hierarchy as needed. This means that the OS must propagate router abstraction level calls to **updateClassifier** through the processor hierarchy to the master processor where they can be fed into the classifier at the top level.

2.3.2 Thread Assignment and Scheduling

The OS provides an abstraction of computation in the form of a thread API. At the router abstraction level, each forwarder and output scheduler runs as a separate thread on a particular processor. As mentioned above, classifiers are multithreaded with the first level of classification running on the processor which owns the input port and the last level of classification running on the master processor. The output scheduler threads run on the processor that owns the output port. Forwarder threads and intermediate classification threads can run on *any* processor in the router. Intermediate classification threads are statically assigned to processors at initialization time.

In Figure 4 we saw an example of the **createPath** instantiating a new path. An important part of path creation is identifying on which processor to instantiate the forwarder. The OS supports this decision by maintaining a database of resource reservations for processor cycles and internal switching cycles. In addition to the traditional object code, a *forwarder object file* also contains per packet processing and switching requirements for each processor architecture supported by the forwarder object file. During a **createPath** call, the OS uses the resource database to determine a list of eligible processors that have enough processing cycles to run the function. With the internal router graph, the OS attempts to find a path that has enough switching cycles to support the request. Paths are greedily considered from shortest to longest. If no path is found which can support the request, the call to **createPath** fails.

After new forwarder threads are instantiated, they must be scheduled along with all the other classifier, forwarder, and output scheduler threads. Because our architecture supports a heterogeneous distributed processing model, the OS must provide support for scheduling computations across the entire router. In

Section 2.2.2 we defined a *scheduling domain* as a tightly bound set of processors (usually, a single processor) running a single thread scheduler. Because the amount of processing required on each packet is small, we have chosen not to have a single, fine-grained scheduler for all the threads of the router. Instead, each processor runs a completely independent scheduler. The master scheduler (running on the master processor) provides only coarse grain adjustments to each scheduling domain to guide their independent scheduling decisions.

The scheduler running in each scheduling domain builds upon our existing scheduling work [45], which in turn, is derived from the WF^{2}Q+ scheduler [5]. We have added hooks to support coarse grain influence from the master processor. In brief, the scheduler assigns a processor share to each forwarder according the the *F_parms* passed to the **createPath** operation. Processor shares are also assigned to the scheduler and classifier threads. The share-based scheduler is augmented to use information about the states of the queues to determine when a forwarder or output scheduler thread should run. In the case of an output scheduler, packets in the queue are assumed to be in local memory so that when an output scheduler thread is activated it will not immediately block attempting to fetch the packet data from a remote processor. This helps keep the output link full. We explore the issue of ensuring that the packet data is local to the output scheduler in the next section.

In Section 2.1 we stated several design rules, including: (1) classifiers must completely classify a packet and not modify the packet, (2) forwarders must have a single input and a single output, and (3) output schedulers do not modify packets. These design rules are motivated by the need to support QoS in our router. When a QoS reservation is made, certain resources are reserved for the corresponding path. The thread scheduler must ensure that the threads belonging to particular QoS paths get the share of the processor they need to meet their reservation. Because the classification threads must be scheduled based on link bandwidth (because the flow for a packet is unknown until *after* classification), we want the classifiers to perform the minimum amount of processing. (This allows the thread scheduler to have control over the maximum amount of processing.) Once packets are classified, forwarder threads can be scheduled based on the QoS reserved for the corresponding flow. Our design stipulates that output schedulers not modify packets. This allows the scheduler to make its thread scheduling decision based on the state of the queues going into and coming from the output scheduler without needing to estimate the amount of processing which might need to be performed on the packet. Because no packet processing occurs in the classifier or output scheduler, all the processing must occur in the forwarder. We have chosen not to perform link-layer processing in the output scheduler, the forwarder can perform arbitrary processing at the link layer. The downside is that if there are N different media types in the router, each processor must be capable of handling all N link layers. However, we expect N to be small enough that $N \times N$ translation will not be an issue.

2.3.3 Internal Packet Routing

It is well known that routers should internally copy data as little as possible. Our architecture helps reduce the amount of data copying by sending an *internal routing header* (rather than the entire packet) from place to place. This internal routing header contains the initial bytes of the packet plus an ordered scatter/gather list of pointers to blocks containing the data of the packet. A reference count is associated with each block; the OS uses this count to determine when a block can be recycled. (The classifier changes the reference count from its nominal value of one when a multicast or broadcast packet is detected; the reference count is set to the number of destinations.)

An interesting property of IP routers is that in most cases only the header need be modified by the forwarder; the body/payload of the packet remains unchanged. When activated by the thread scheduler, a

forwarder first reads the next internal routing header from its input queue, fetches any (remote) packet data it needs and then performs the computation. After processing the packet, the forwarder sends the updated internal routing header to its output queue (which is connected to the appropriate output scheduler). It is the responsibility of the queue API to make sure all the packet data is local to the output scheduler before the internal routing header is placed on the queue. Because the classification hierarchy and the forwarder on the invoked path may not have needed the entire packet to complete the classification and requisite forwarding function, the packet may be scattered in several blocks across several processors. Anytime a block is moved, the internal routing header is updated with the new location. When the internal routing header reaches the output scheduler's processor, the data must be copied to local memory before the internal routing header can be placed on the output scheduler's inbound queue. Figure 11 illustrates the sequence.

1. The forwarder, *F*, send an internal routing header (IRH) to the output scheduler, *S*.
2. The *queue server (QS)*, preprocesses the IRH to determine the location of the packet data.
3. The QS uses the HAL data movement primitives to fetch the packet data.
4. The QS modifies the IRH to indicate that the data is now local and places it on the input queue where it is visible to *S*.
5. *S* reads the IRH from the queue.
6. *S* directs the data from local memory to the device queue.

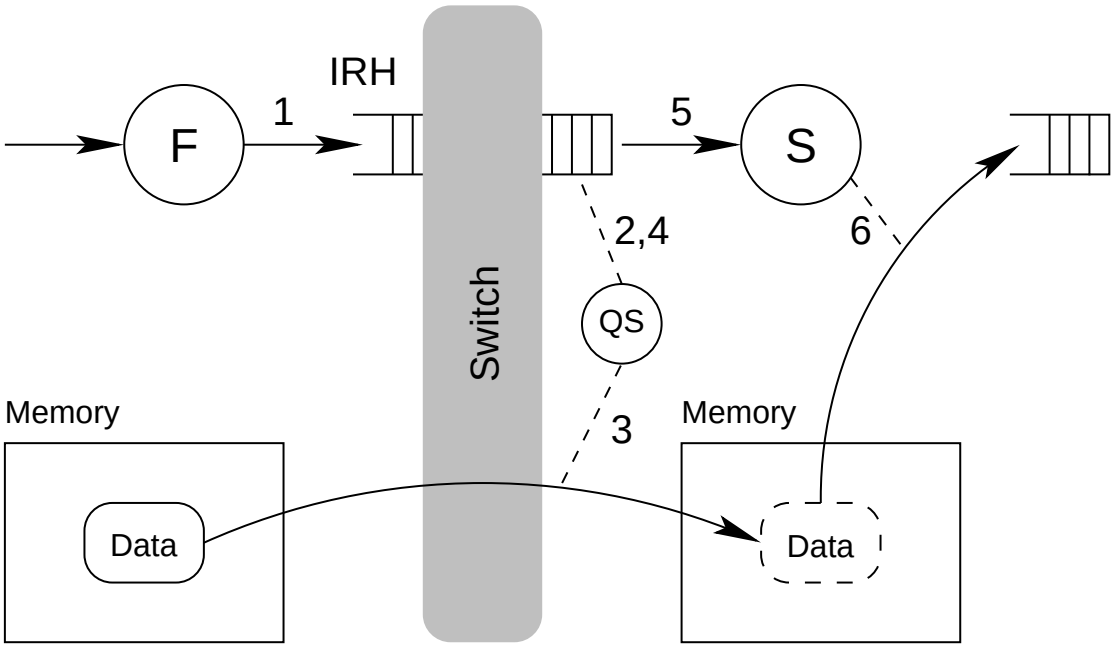


Figure 11: This shows the steps performed by the OS when an internal routing header (IRH) is sent from a forwarder, *F*, on one processor to an output scheduler, *S*, on another processor. (See text.)

Until now we have discussed the issues with moving the data through the router. Eventually, these buffers must be recycled when the packet data is no longer needed. The HAL data movement commands are really data *copying* commands. It is the responsibility of the OS to manage the buffers. When we wish to *move* a buffer referenced by an IRH, we send a release message to the source processor. The processor decrements the reference count associated with the block(s) and reuses any whose count has reached zero.

2.3.4 Queues

Recall that our queuing system is modeled after I_2O. This means that we are pushing and popping *pointers* to IRHs. In order to effect the transfer of an IRH, we use two hardware-level queues to implement the high-level queues of the router abstraction. To simplify the explanation of the process, we will temporarily refer to the low-level queues as (circular) buffers. One buffer contains pointers to empty IRH frames, and one buffer contains pointers to to-be-processed IRH frames. Putting an IRH onto a queue involves first pulling a pointer to a free frame from the free-frame buffer, filling the frame using the data movement primitives in the HAL, and then placing the pointer on the to-be-processed buffer. Getting an IRH from a queue involves first retrieving a pointer from the to-be-processed buffer, processing the data, and then returning the block to the free pool by placing its pointer on the free-block buffer. We elect this method because I_2O gives us hardware support for managing these circular buffers and we want to take advantage of this support when it is available.

With the path of each flow requiring two high-level queues (one of which will usually span a switching element) and the router handling 1000's of flows, the OS must be able to efficiently create and destroy queues. Because the limitation imposed by the hardware (Section 2.2.4) will generally be much less than the number of queues the router must support, the OS must use the limited hardware to create a larger number of virtual queues. These virtual queues are multiplexed on top of the hardware queues.

When a virtual queue is established, it is given an identification key. Each queue element is tagged with the key of the destination virtual queue. A demultiplexing step occurs on the receiving end to assign the incoming item to the correct queue. Note that this step can be combined with the queue server thread of Figure 11.

2.4 Realization in Linux

We have implemented a Linux kernel module which is a unified device driver for both the PMC694 and the IXP1200 EEB. (By choosing to develop a Linux kernel module, our device driver will also work with the communication-oriented operating system, Scout [35].) Figure 12 shows the driver instantiated with one IXP1200 EEB and two PMC694 boards. In this router configuration, there are a total of twelve 100 Mbit/s Ethernet ports.

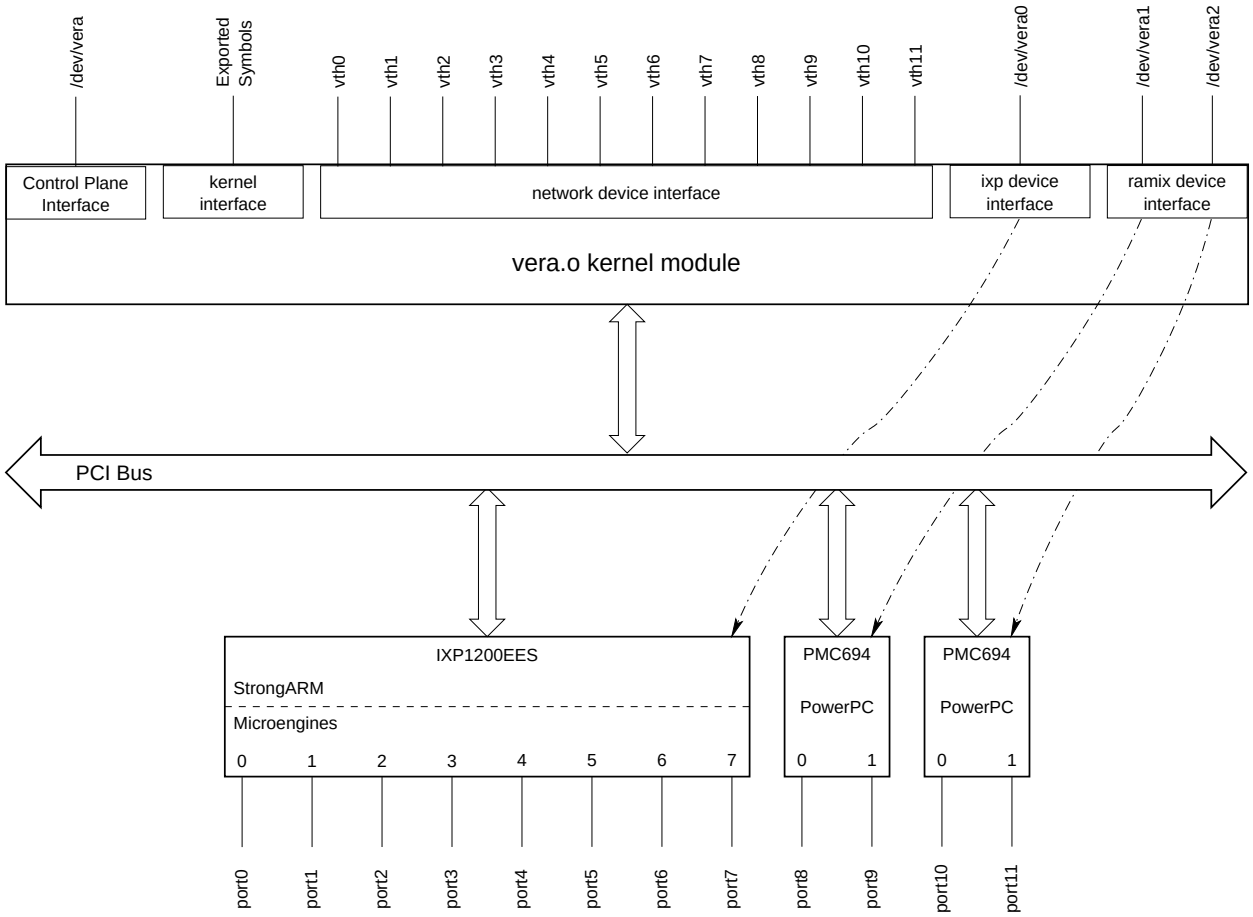


Figure 12: The **vera.o** kernel module. In this figure, the module has been instantiated in a system containing one IXP1200 EEB and two PMC694 boards.

As shown in Figure 12, there are four types of interfaces to the driver:

Control Plane Interface: The driver exports a character device interface as **/dev/vera**. This is used to perform updates (from user space) which do not apply to a specific hardware device or network interface. (For example, routing table updates made via the classification hierarchy as discussed in Section 2.1.1.)

Kernel Interface: The module exports symbols which allow kernel modules which are dynamically loaded (after **vera.o** is loaded) to extend the **ioctl** interface to either **/dev/vera** or **/dev/vera*N***.

Virtualized Network Interfaces: A key element of the module is that the physical network interfaces are *virtualized*. Because we want the intelligent NICs to route the packets directly between hardware devices (either within a board or from board to board), many packets will never arrive at the network driver interface to the Linux kernel. However, packets which do arrive on a particular port and which are to be handled by the kernel are sent to their corresponding virtual interface. In this way, packets which are not handled by VERA can be processed normally by the Linux kernel.

Device Interfaces: When the module is instantiated, a character device of the form **/dev/vera*N*** is assigned to each PMC694 or IXP1200 EEB device. This interface allows the boards to be initialized, code to be downloaded, and gives access to the memory and registers of each board.

3 Global Resources

Moving from a localized view to a more global perspective, we observe that the availability of information services is negatively impacted by overloaded servers and congested networks. To alleviate these bottlenecks, it is becoming increasingly common to construct network services using redundant resources, so-called Content Distribution Networks (CDN) [1, 18, 33]. CDNs deploy geographically-dispersed server surrogates and distribute client requests to an "appropriate" server based on various considerations.

CDNs are designed to improve two performance metrics: *response time* and *system throughput*. Response time, usually reported as a cumulative distribution of latencies, is of obvious importance to clients, and represents the primary marketing case for CDNs. System throughput, the average number of requests that can be satisfied each second, is primarily an issue when the system is heavily loaded, for example, when a flash crowd is accessing a small set of pages, or a Distributed Denial of Service (DDoS) attacker is targeting a particular site [20]. System throughput represents the overall robustness of the system since either a flash crowd or a DDoS attack can make portions of the information space inaccessible.

Given a sufficiently widespread distribution of servers, CDNs use several, sometimes conflicting, factors to decide how to distribute client requests. For example, to minimize response time, a server might be selected based on its *network proximity*. In contrast, to improve the overall system throughput, it is desirable to evenly *balance* the load across a set of servers. Both throughput and response time are improved if the distribution mechanism takes *locality* into consideration by selecting a server that is likely to already have the page being requested in its cache.

Although the exact combination of factors employed by commercial systems is not clearly defined in the literature, evidence suggests that the scale is tipped in favor of reducing response time. This effort addressed the problem of designing a request distribution mechanism that is both responsive across a wide range of loads, and robust in the face of flash crowds and DDoS attacks. Specifically, our main contribution is to explore the design space of strategies employed by the request redirectors, and to define a class of new algorithms that carefully balance load, locality, and proximity. We use large-scale detailed simulations to evaluate the various strategies. These simulations clearly demonstrate the effectiveness of our new algorithms: they produce a 60-91% improvement in system capacity when compared with published information about commercial CDN technology, user-perceived response latency remains low, and the system scales well with the number of servers.

3.1 Building Blocks

The idea of a CDN is to geographically distribute a collection of *server surrogates* that cache pages normally maintained in some set of *backend servers*. Thus, rather than let every client try to connect to the original server, it is possible to spread request load across many servers. Moreover, if a server surrogate happens to reside close to the client, the client's request could be served without having to cross a long network path. In this effort, we observe this general model of a CDN, and assume any of the server surrogates can serve any request on behalf of the original server. Where to place these surrogates, and how to keep their contents up-to-date, has been addressed by other CDN research [1, 18, 33]. Here, we make no particular assumptions about servers' strategic locations.

Besides a large set of servers, CDNs also need to provide a set of *request redirectors*, which are middleware entities that forward client requests to appropriate servers based on one of the strategies described in the next section. To help understand these strategies, this section first outlines various mechanisms that could be employed to implement redirectors, and then presents a set of hashing schemes that are at the heart

of redirection.

3.1.1 Redirector Mechanisms

Several mechanisms can be used to redirect requests [4], including augmented DNS servers, HTTP-based redirects, and smart intermediaries such as routers or proxies.

A popular redirection mechanism used by current CDNs is to augment DNS servers to return different server addresses to clients. Without URL rewriting that changes embedded objects to point to different servers, this approach has site-level granularity, while schemes that rewrite URLs can use finer granularity and thus spread load more evenly. Client-side caching of DNS mappings can be avoided using short expiration times.

Servers can perform the redirection process themselves by employing the HTTP "redirect" response. However, this approach incurs an additional round-trip time, and leaves the servers vulnerable to overload by the redirection task itself. Server bandwidth is also consumed by this process.

The redirection function can also be distributed across intermediate nodes of the network, such as routers or proxies. These redirectors either rewrite the outbound requests, or send HTTP redirect messages back to the client. If the client is not using explicit (forward mode) proxying, then the redirectors must be placed at choke points to ensure all traffic is handled. Placing proxies closer to the edge yields well-confined easily-identifiable client populations, while moving them closer to the server can result in more accurate feedback and load information.

To allow us to focus on redirection strategies and to reduce the complexity of considering the various combinations outlined in this section, we make the following assumptions: redirectors are located at the edge of a client site, they receive the full list of cooperating servers through DNS or some other out-of-band communication, they rewrite outbound requests to pick the appropriate server, and they passively learn approximate server load information by observing client communications. We do not rely on any centralization, and all redirectors operate independently. These assumptions—in particular, the imperfect information about server load—do not have a significant impact on the results.

3.1.2 Hashing Schemes

Our geographically dispersed redirectors cannot easily adapt the request routing schemes suited for more tightly-coupled LAN environments [22, 40], since the latter can easily obtain instantaneous state about the entire system. Instead, we construct strategies that use hashing to deterministically map URLs into a small range of values. The main benefit of this approach is that it eliminates inter-redirector communication, since the same output is produced regardless of which redirector receives the URL. The second benefit is that the range of resulting hash values can be controlled, trading precision for the amount of memory consumed by bookkeeping.

The choice of which hashing style to use is one component of the design space, and is somewhat flexible. The various hashing schemes have some impact on computational time and request reassignment behavior on node failure/overload. However, as we discuss in the next section, the computational requirements of the various schemes can be reduced by caching.

Modulo Hashing – In this "classic" approach, the URL is hashed to a number modulo the number of cooperating servers. While this approach is computationally efficient, it is unsuitable because the modulus changes when the server set changes, causing most documents to change server assignments. While we do not expect frequent changes in the set of servers, the fact that the addition of new servers into the set will cause massive reassignment is undesirable.

Category	Strategy	Hashing Scheme	Dynamic Server Set	Load Aware
Random	Random			No
Static	R-CHash	CHash	No	No
	R-HRW	HRW	No	No
Static +Load	LR-CHash	CHash	No	Yes
	LR-HRW	HRW	No	Yes
Dynamic	CDR	HRW	Yes	Yes
	FDR	HRW	Yes	Yes
	FDR-Global	HRW	Yes	Yes
Network Proximity	NP-CHash	CHash	No	No
	NPLR-CHash	CHash	No	Yes
	NP-FDR	HRW	Yes	Yes

Table 3: Properties of Request Redirection Strategies

Consistent Hashing [29, 30] – In this approach, the URL is hashed to a number in a large, circular space, as are the names of the cooperating servers. The URL is assigned to the server that lies closest on the circle to its hash value. A search tree can be used to reduce the search to logarithmic time. If a node fails in this scheme, its load shifts to its neighbors, so the addition/removal of a server only causes local changes in request assignments.

Highest Random Weight [53] – This approach is the basis for CARP [9], and consists of generating a list of hash values by hashing the URL and each server's name and sorting the results. Each URL then has a deterministic order to access the set of servers, and this list is traversed until a suitably-loaded server is found. This approach requires more computation than Consistent Hashing, but has the benefit that each URL has a different server order, so a node failure results in the remaining nodes evenly sharing the load. To reduce computation cost, the top few entries for each hash value can be cached.

3.2 Strategies

This section explores the design space for the request redirection strategies. As a quick reference, we summarize the properties of the different redirection algorithms in Table 3, where the strategies are categorized based on how they address locality, load and proximity.

The first category, Random, contains a single strategy, and is used primarily as a baseline. We then discuss four static algorithms, in which each URL is mapped onto a fixed set of server replicas—the Static category includes two schemes based on the best-known published algorithms, and the Static+Load category contains two variants that are aware of each replica's load. The four algorithms in these two static categories pay increasing attention to locality. Next, we introduce two new algorithms—denoted **CDR** and **FDR**—that factor both load and locality into their decision, and each URL is mapped onto a dynamic set of server replicas. We call this the Dynamic category. Finally, we factor network proximity into the equation, and present another new algorithm—denoted **NP-FDR**—that considers all aspects of network proximity, server locality, and load.

3.2.1 Random

In the random policy, each request is randomly sent to one of the cooperating servers. We use this scheme as a baseline to determine a reasonable level of performance, since we expect the approach to scale with the

number of servers and to not exhibit any pathological behavior due to patterns in the assignment. It has the drawback that adding more servers does not reduce the working set of each server. Since serving requests from main memory is faster than disk access, this approach is at a disadvantage versus schemes that exploit URL locality.

3.2.2 Static Server Set

We now consider a set of strategies that assign a fixed number of server replicas to each URL. This has the effect of improving locality over the Random strategy.

Replicated Consistent Hashing. In the Replicated Consistent Hashing (**R-CHash**) strategy, each URL is assigned to a set of replicated servers. The number of replicas is fixed, but configurable. The URL is hashed to a value in the circular space, and the replicas are evenly spaced starting from this original point. On each request, the redirector randomly assigns the request to one of the replicas for the URL. This strategy is intended to model the mechanism used in published content distribution networks, and is virtually identical[1] to the scheme described in [29] and [30].

Replicated Highest Random Weight. The Replicated Highest Random Weight (**R-HRW**) strategy is the counterpart to R-CHash, but with a different hashing scheme used to determine the replicas. To the best of our knowledge, this approach is not used in any existing content distribution network. In this approach, the set of replicas for each URL is determined by using the top N servers from the ordered list generated by Highest Random Weight hashing. Versus R-CHash, this scheme is less likely to generate the same set of replicas for two different URLs. As a result, the less-popular URLs that may have some overlapping servers with popular URLs are also likely to have some other less-loaded nodes in their replica sets.

3.2.3 Load-Aware Static Server Set

The Static Server Set schemes randomly distribute requests across a set of replicas, which shares the load but without any active monitoring. We extend these schemes by introducing load-aware variants of these approaches. To perform fine-grained load balancing, these schemes maintain local estimates of server load in the redirectors, and use this information to pick the least-loaded member of the server set. The load-balanced variant of R-CHash is called **LR-CHash**, while the counterpart for R-HRW is called **LR-HRW**.

3.2.4 Dynamic Server Set

We now consider a new category of algorithms that dynamically adjust the number of replicas used for each URL in an attempt to maintain both good server locality and load balancing. By reducing unnecessary replication, the working set of each server is reduced, resulting in better file system caching behavior.

Coarse Dynamic Replication. Coarse Dynamic Replication (**CDR**) adjusts the number of replicas used by redirectors in response to server load and demand for each URL. Like R-HRW, CDR uses HRW hashing

[1]The scheme described in these papers also includes a mechanism to use coarse-grained load balancing via virtual server names. When server overload is detected, the corresponding content is replicated across all servers in the region, and the degree of replication shrinks over time. However, the schemes are not described in enough detail to replicate.

to generate an ordered list of servers. Rather than using a fixed number of replicas, however, the request target is chosen using coarse-grained server load information to select the first "available" server on the list.

Figure 13 shows how a request redirector picks the destination server for each request. This decision process is done at each redirector independently, using the load status of the possible servers. Instead of relying on heavy communications between servers and request redirectors to get server load status, we use local load information observed by each redirector as an approximation. We currently use the number of active connections to infer the load level, but we can also combine this information with response latency, bandwidth consumption, etc.

```
find_server(url, S) {
   foreach server s_i in server set S,
      weight_i = hash(url, address(s_i));
   sort weight;
   foreach server s_j in decreasing order of weight_j {
      if satisfy_load_criteria(s_j) then {
         targetServer ← s_j;
         stop search;
      }
   }
   if targetServer is not valid then
      targetServer ← server with highest weight;
   route request url to targetServer;
}
```

Figure 13: Coarse Dynamic Replication

As the load increases, this scheme changes from using only the first server on the sorted list to spreading requests across several servers. Some documents normally handled by "busy" servers will also start being handled by less busy servers. Since this process is based on aggregate server load rather than the popularity of individual documents, servers hosting some popular documents may find more servers sharing their load than servers hosting collectively unpopular documents. In the process, some unpopular documents will be replicated in the system simply because they happen to be primarily hosted on busy servers. At the same time, if some documents become extremely popular, it is conceivable that all of the servers in the system could be responsible for serving them.

Fine Dynamic Replication. A second dynamic algorithm—Fine Dynamic Replication (**FDR**)—addresses the problem of unnecessary replication in CDR by keeping information on URL popularity and using it to more precisely adjust the number of replicas. By controlling the replication process, the per-server working sets should be reduced, leading to better server locality, and thereby better response time and throughput.

The introduction of finer-grained bookkeeping is an attempt to counter the possibility of a "ripple effect" in CDR, which could gradually reduce the system to round-robin under heavy load. In this scenario, a very popular URL causes its primary server to become overloaded, causing extra load on other machines. Those machines, in turn, also become overloaded, causing documents destined for them to be served by their secondary servers. Under heavy load, it is conceivable that this displacement process ripples through the system, reducing or eliminating the intended locality benefits of this approach.

```
find_server(url, S) {
   walk_entry ← walkLenHash(url);
   w_len ← walk_entry.length;
   foreach server s_i in server set S,
      weight_i = hash(url, address(s_i));
   sort weight;
   s_candidate ← least-loaded server of top w_len servers;
   if satisfy_load_criteria(s_candidate) then {
     targetServer ← s_candidate;
     if (w_len > 1 &&
         timenow() - walk_entry.lastUpd > changeThresh)
        walk_entry.length −−;
   } else {
      foreach remaining server s_j in decreasing weight order {
         if satisfy_load_criteria(s_j) then {
            targetServer ← s_j;
            stop search;
         }
      }
      walk_entry.length ← actual search steps;
   }
   if walk_entry.length changed then
      walk_entry.lastUpd ← timenow();
   if targetServer is not valid then
      targetServer ← server with highest weight;
   route request url to targetServer;
}
```

Figure 14: Fine Dynamic Replication

To reduce extra replication, FDR keeps an auxiliary structure at each redirector that maps each URL to a "walk length," indicating how many servers in the HRW list should be used for this URL. Using a minimum walk length of one provides minimal replication for most URLs, while using a higher minimum will always distribute URLs over multiple servers. When the redirector receives a request, it uses the current walk length for the URL and picks the least-loaded server from the current set. If even this server is busy, the walk length is increased and the least-loaded server is used.

This approach tries to keep popular URLs from overloading servers and displacing less-popular objects in the process. The size of the auxiliary structure is capped by hashing the URL into a range in the thousands to millions. Hash collisions may cause some URLs to have their replication policies affected by popular URLs. As long as the the number of hash values exceeds the number of servers, the granularity will be significantly better than the Coarse Dynamic Replication approach. The redirector logic for this approach is shown in Figure 14. To handle URLs that become less popular over time, with each walk length, we also keep the time of its last modification. We decrease the walk length if it has not changed in some period of time.

As a final note, both dynamic replication approaches require some information about server load, specif-

ically how many outstanding requests can be sent to a server by a redirector before the redirector believes it is busy. We currently allow the redirectors to have 300 outstanding requests per server, at which point the redirector locally decides the server is busy. It would also be possible to calibrate these values using both local and global information—using its own request traffic, the redirector can adjust its view of what constitutes heavy load, and it can perform opportunistic communication with other redirectors to see what sort of collective loads are being generated. The count of outstanding requests already has some feedback, in the sense that if a server becomes slow due to its resources (CPU, disk, bandwidth, etc.) being stressed, it will respond more slowly, increasing the number of outstanding connections. To account for the inaccuracy of local approximation of server load at each redirector, in our evaluations, we also include a reference strategy, **FDR-Global**, where all redirectors have perfect knowledge of the load at all servers.

Conceivably, Consistent Hashing could also be used to implement CDR and FDR. We tested a CHash-based CDR and FDR, but they suffer from the "ripple effect" and sometimes yield even worse performance than load-aware static replication schemes. Part of the reason is that in Consistent Hashing, considering the fact that servers are mapped onto a circular space, the relative order of servers for each URL will be *effectively* the same. This means the load migration will take an uniform pattern; and the less-popular URLs that may have overlapping servers with popular URLs are unlikely to have some other less-loaded nodes in their replica sets. Therefore, we only present CDR and FDR based on HRW.

3.2.5 Network Proximity

Many commercial CDNs start server selection with network proximity matching. For instance, [29] indicates that CDN's hierarchical authoritative DNS servers can map client's (actually its local DNS server's) IP address to a geographic region within a particular network and then combine it with network and server load information to select a server. Other research [28] shows that in practice, CDNs succeed not by always choosing the "optimal" server, but by avoiding notably bad servers.

For the sake of studying system capacity, we make a conservative simplicifaction by treating the entire network topology as a single geographic region. This is because taking network proximity into account generally reduces system capacity in order to improve response time.

However, to see the affect of including proximity in server selection, we introduce three strategies that explicitly factor network proximity into the decision. Our redirector measures servers' geographical/topological location information through *ping*, *traceroute* or similiar mechanisms and uses this information to calculate an "effective load" when choosing servers.

To calculate the *effective load*, redirectors multiply the raw load metric with a normalized *standard distance* between the redirector and the server. Redirectors gather distances to servers using round trip time (RTT), routing hops, or similar information. These raw distances are normalized by dividing by the minimum locally-observed distance, yielding the standard distance. In our simulations, we use RTT for calculating raw distances.

FDR with Network Proximity (NP-FDR) is the counterpart of FDR, but it uses effective load rather than raw load. Similarly, **NPLR-CHash** is the proximity-aware version of LR-CHash. The third strategy, **NP-CHash**, adds network proximity to the load-oblivious R-CHash approach by assigning requests such that each surrogate in the fixed-size server set of a URL will get a share of total requests for that URL inversely proportional to the surrogate's distance from the redirector. As a result, closer servers in the set get a larger share of the load.

The use of effective load biases server selection in favor of closer servers when raw load values are comparable. For example, in standard FDR, raw load values reflect the fact that distant servers generate

replies more slowly, so some implicit biasing exists. However, by explicitly factoring in proximity, NP-FDR attempts to reduce global resource consumption by favoring shorter network journeys.

Although we currently calculate effective load this way, other options exist. For example, effective load can take other dynamic load/proximity metrics into account, such as network congestion status through real time measurement, thereby reflecting instantaneous load conditions.

3.3 Evaluation Methodology

The goal of this work is to examine how these strategies respond under different loads, and especially how robust they are in the face of flash crowds and other abnormal workloads that might be used for a DDoS attack. Attacks may take the form of legitimate traffic, making them difficult to distinguish from flash crowds.

Evaluating the various algorithms described in Section 3.2 on the Internet is not practical, both due to the scale of the experiment required and the impact a flash crowd or attack is likely to have on regular users. Simulation is clearly the only option. Unfortunately, there has not been (up to this point) a simulator that considers both network traffic and server load. Existing simulators either focus on the network, assuming a constant processing cost at the server, or they accurately model server processing (including the cache replacement strategy), but use a static estimate for the network transfer time. In the situations that interest us, both the network and the server are important.

To remedy this situation, we developed a new simulator that combines network-level simulation with OS/server simulation. Specifically, we combine the NS simulator with Logsim, allowing us to simulate network bottlenecks, round-trip delays, and OS/server performance. NS-2 [37] is a packet-level simulator that has been widely-used to test TCP implementations. However, it does not simulate much server-side behavior. Logsim is a server cluster simulator used in previous research on LARD [40], and it provides detailed and accurate simulation of server CPU processing, memory usage, and disk access. This section describes how we combine these two simulators, and discusses how we configure the resulting simulator to study the algorithms presented in Section 3.2.

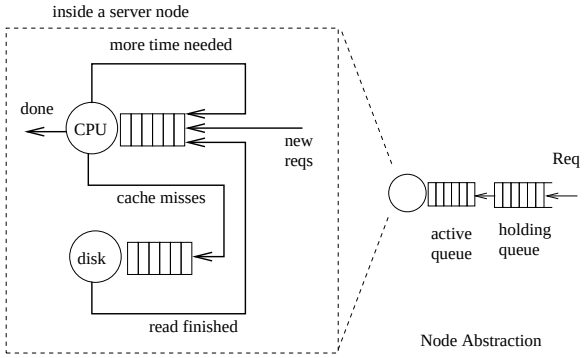


Figure 15: Logsim Simulator

3.3.1 Simulator

A model of Logsim is shown in Figure 15. Each server node consists of a CPU and locally attached disk(s), with separate queues for each. At the same time, each server node maintains its own memory cache of a

configurable size and replacement policy. Incoming requests are first put into the holding queue, and then moved to the active queue. The active queue models the parallelism of the server, for example, in multiple process or thread server systems, the maximum number of processes or threads allowed on each server.

We combined Logsim with NS-2 as follows. We keep NS-2's event engine as the main event manager, wrap each Logsim event as a NS-2 event, and insert it into the NS-2 event queue. All the callback functions are kept unchanged in Logsim. When crossing the boundary between the two simulators, tokens (continuations) are used to carry side-specific information. To speed up the simulation time, we also re-implemented several NS-2 modules and performed other optimizations.

On the NS side, all packets are stored and forwarded, as in a real network, and we use two-way TCP. We currently use static routing within NS-2, although we may run simulations with dynamic routing in the future.

On the Logsim side, the costs for the basic request processing were derived by performing measurements on a 300MHz Pentium II machine running FreeBSD 2.2.5 and the Flash web server [39]. Connection establishment and tear-down costs are set at 145μs, while transmit processing incurs 40μs per 512 bytes. Using these numbers, an 8-KByte document can be served from the main memory cache at a rate of approximately 1075 requests/sec. When disk access is needed, reading a file from the disk has a latency of 28ms. The disk transfer time is 410μs per 4 KBytes. For files larger than 44 KBytes, and additional 14ms is charged for every 44 KBytes of file length in excess of 44 KBytes. The replacement policy used on the servers is Greedy-Dual-Size (GDS)[6], as it appears to be the best known policy for Web workloads. 32MB memory is available for caching documents on each server and every server node has one disk. This server is intentionally slower than the current state-of-the-art (it is able to service approximately 600 requests per second), but this allows the simulation to scale to a larger number of nodes.

The final simulations are very heavy-weight, with over a thousand nodes and a very high aggregate request rate. We run the simulator on a 4-processor/667MHz Alpha with 8GB RAM. Each simulation requires 2-6GB of RAM, and generally takes 20-50 hours of wall-clock time.

3.3.2 Network Topology

It is not easy to find a topology that is both realistic and makes the simulation manageable. We choose to use a slightly modified version the NSFNET backbone network T3 topology, as shown in Figure 16.

In this topology, the round-cornered boxes represent backbone routers with the approximate geographical location label on it. The circles, tagged as R1, R2..., are regional routers;[2] small circles with "C" stand for client hosts; and shaded circles with "S" are the cooperating servers. In the particular configuration shown in the figure, we put 64 cooperating servers behind regional routers R0, R1, R7, R8, R9, R10, R15, R19, where each router sits in front of 8 servers. We distribute 1,000 client hosts evenly behind the other regional routers, yielding a topology of nearly 1,100 nodes. The redirector algorithms run on the regional routers that sit in front of the clients.

The latencies of servers to regional routers are set randomly between 1ms to 3ms; those of clients to regional routers are between 5 ms and 20ms; those of regional routers to backbone routers are between 1 to 10ms; latencies between backbone routers are set roughly according to their geographical distances, ranging from 8ms to 28ms.

To simulate high request volume, we deliberately provision the network with high link bandwidth by setting the backbone links at 2,488Mbps, and links between regional routers and backbone routers at 622Mbps. Links between servers and regional routers are 100Mbps and those between clients and their regional servers

[2]These can also be thought of as edge/site routers, or the boundary to an autonomous system

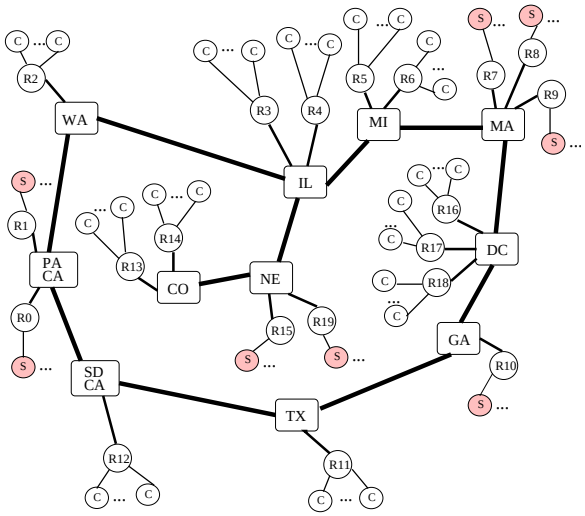


Figure 16: Network Topology

are randomly between 10Mbps and 45Mbps. All the queues at routers are drop tail, with the backbone routers having room to buffer 1024 packets, and all other routers able to buffer 512 packets.

3.3.3 Workload and Stability

We determine system capacity using a trace-driven simulation and gradually increase the aggregate request rate until the system fails. We use a two month trace of server logs obtained at Rice University, which contains 2.3 million requests for 37,703 files with a total size of 1,418MB [40], and has properties similar to other published traces.

The simulation starts with the clients sharing the trace and sending requests at a low aggregate rate in an open-queue model. Each client gets the name of the document sequetially from the shared trace when it needs to send a request, and timing information in the trace is ignored. The request rate is increased by 1% every simulated six seconds, regardless of whether previous requests have completed. This approach gradually warms the server memory caches and drives the servers to their limits over time. We configure Logsim to handle at most 512 simultaneous requests and queue the rest. The simulation is terminated when the offered load overwhelms the servers.

Flash crowds, or DDoS attacks in bursty legitimate traffic form, are simulated by randomly selecting some clients as *intensive* requesters and randomly picking a certain number of hot-spot documents. These intensive requesters randomly request the hot documents at the same rate as normal clients, making them look no different than other legitimate users. We believe that this random distribution of intensive requesters and hot documents is a quite general assumption since we do not require any special detection or manual intervention to signal the start of a flash crowd or DDoS attack.

We define a server as being overloaded when it can no longer satisfy the rate of incoming requests and is unlikely to recover in the future. This approach is designed to determine when service is actually being denied to clients, and to ignore any short-term behavior which may be only undesirable, rather than fatal. Through experimentation, we find that when a server's request queue grows beyond 4 to 5 times the number of simultaneous connections it can handle, throughput drops and the server is unlikely to recover.

Thus, we define the threshold for a server *failure* to be when the request queue length exceeds five times the simultaneous connection parameter. Since we increase the offered load 1% every 6 seconds, we record the request load exactly 30 seconds before the first server fails, and declare this to be the system's maximum capacity.

Although we regard any single server failure as a system failure in our simulation, the strategies we evaluate all exhibit similar behavior—significant numbers of servers fail at the same time, implying that our approach to deciding system capacity is not biased toward any particular scheme.

3.4 Results

This section evaluates how the different strategies in Table 3 perform, both under normal conditions and under flash crowds or DDoS attacks. Network proximity and other factors that affect the performance of these strategies are also addressed.

3.4.1 Normal Workload

Before evaluating these strategies under flash crowds or other attack, we first measure their behavior under normal workloads. In these simulations, all clients generate traffic similar to normal users and gradually increase their request rates as discussed in Section 3.3.3. We compare aggregate system capacity and user-perceived latency under the different strategies, using the topology shown in Figure 16.

Optimal Static Replication. The static replication schemes (R-CHash, R-HRW, and their variants) use a configurable (but fixed) number of replicas, and this parameter's value influences their performance. Using a single replica per URL perfectly partitions the file set, but can lead to early failure of servers hosting popular URLs. Using as many replicas as available servers degenerates to the Random strategy. To determine an appropriate value, we varied this parameter between 2 and 64 replicas for R-CHash when there are 64 servers available. Increasing the number of replicas per URL initially helps to improve the system's throughput as the load gets more evenly distributed. Beyond a certain point, throughput starts decreasing due to the fact that each server is presented with a larger working set, causing more disk activity. In the 64-server case—the scenario we use throughout the rest of this section—10 server replicas for each URL achieves the optimal system capacity. For all of the remaining experiments, we use this value in the R-CHash and R-HRW schemes and their variants.

System Capacity. The maximum aggregate throughput of the various strategies with 64 servers are shown in Figure 17. Here we do not plot all the strategies and variants, but focus on those impacting throughput substantially. Random shows the lowest throughput at 9,300 req/s before overload. The static replication schemes, R-CHash and R-HRW, outperform Random by 119% and 99%, respectively. Our approximation of static schemes' best behaviors, LR-CHash and LR-HRW, yields 173% better capacity than Random. The dynamic replication schemes, CDR and FDR, show over 250% higher throughput than Random, or more than a 60% improvement over the static approaches and 28% over static schemes with fine-grained load control.

The difference between Random and the static approaches stems from the locality benefits of the hashing in the static schemes. By partitioning the working set, more documents are served from memory by the servers. Note, however, that absolute minimal replication can be detrimental, and in fact, the throughput for only two replicas in Section 3.4.1 is actually lower than the throughput for Random. The difference in

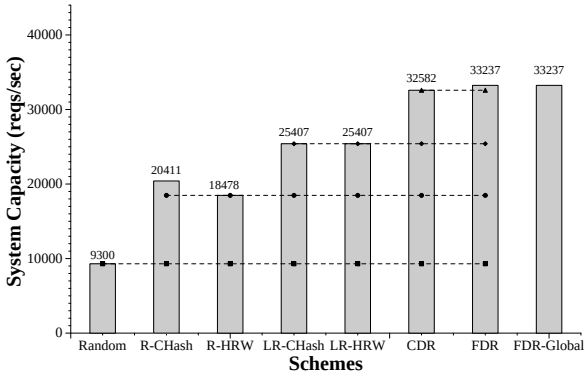


Figure 17: Capacity Comparison under Normal Load

Utilization	*CPU (%)*		*DISK (%)*	
Scheme	Mean	Stddev	Mean	Stddev
Random	21.03	1.36	100.00	0.00
R-CHash	57.88	18.36	99.15	3.89
R-HRW	47.88	15.33	99.74	1.26
LR-CHash	59.48	18.85	97.83	12.51
LR-HRW	58.43	16.56	99.00	5.94
CDR	90.07	11.78	36.10	25.18
FDR	93.86	7.58	33.96	20.38
FDR-Global	91.93	11.81	17.60	15.43

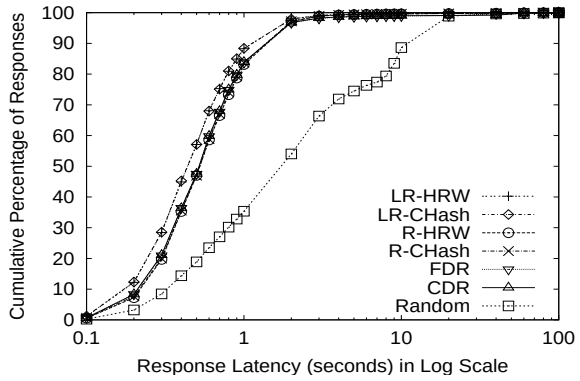
Table 4: Server Resource Utilization at Overload

throughput between R-CHash and R-HRW is 11% in our simulation. However, this difference should not be over emphasized, because changes in the number of servers or workload can cause their relative ordering to change. Considering load helps the static schemes gain about 25% better throughput, but they still do not exceed the dynamic approaches.

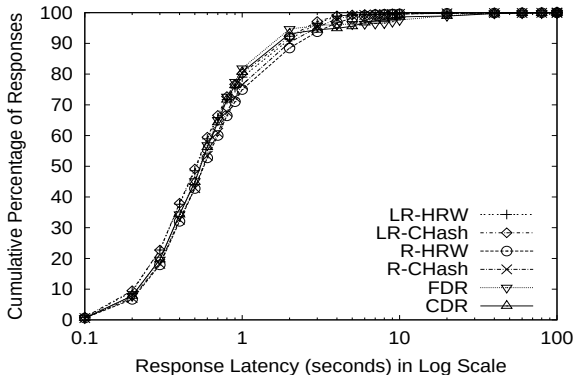
The performance difference between the static (including with load control) and dynamic schemes stems from the adjustment of the number of replicas for the documents. FDR also shows 2% better capacity than CDR.

Interestingly, the difference between our dynamic schemes (with only local knowledge) and the FDR-Global policy (with perfect global knowledge) is minimal. These results suggest that request distribution policies not only fare well with only local information, but that adding more global information may not gain much in system capacity.

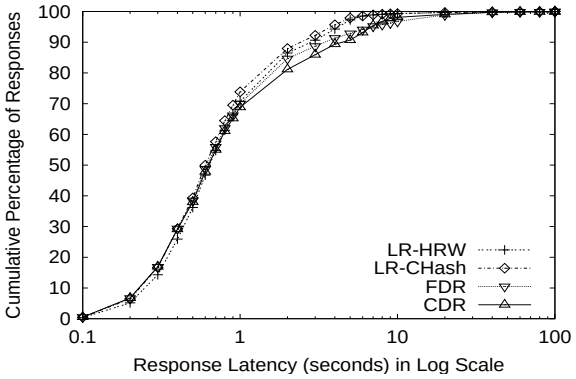
Examination of what ultimately causes overload in these systems reveals that, under normal load, the server's behavior is the factor that determines the performance limit of the system. None of the schemes suffers from saturated network links in these non-attack simulations. For Random, due to the large working set, the disk performance is the limit of the system, and before system failure, the disks exhibit almost 100% activity while the CPU remains largely idle. The R-CHash, R-RHW and LR-CHash and LR-HRW exhibit much lower disk utilization at comparable request rates; but by the time the system becomes overloaded, their bottleneck also becomes the disk and the CPU is roughly 50-60% utilized on average. In the CDR and FDR cases, at system overload, the average CPU is over 90% busy, while most of the disks are only 10-70%



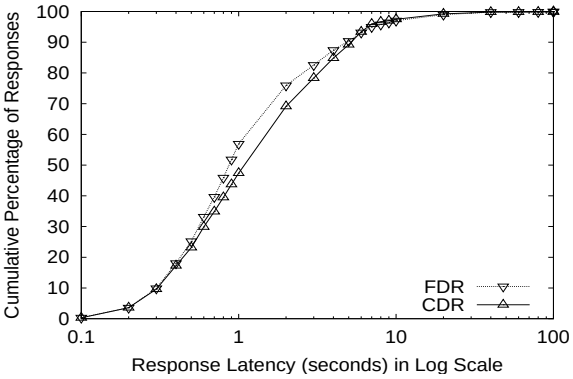
(a) Random limit: 9,300 req/s



(b) R-HRW limit: 18,478 req/s



(c) LR-HRW limit: 25,407 req/s



(d) CDR limit: 32,582 req/s

Figure 18: Response Latency Distribution under Normal Load

utilized. Table 4 summarizes resource utilization of different schemes before server failures (not at the same time point).

These results suggest that the CDR and FDR schemes are the best suited for technology trends, and can most benefit from upgrading server capacities. The throughput of our simulated machines is lower than what can be expected from state-of-the-art machines, but this decision to scale down resources was made to keep the simulation time manageable. *With faster simulated machines, we expect the gap between the dynamic schemes and the others to grow even larger.*

Response Latency. Along with system capacity, the other metric of interest is user-perceived latency, and we find that our schemes also perform well in this regard. To understand the latency behavior of these systems, we use the capacity measurements from Figure 17 and analyze the latency of all of the schemes whenever one category reaches its performance limit. For schemes with similar performance in the same category, we pick the lower limit for the analysis so that we can include numbers for the higher-performing scheme. In all cases, we present the cumulative distribution of all request latencies as well as some statistics

Req Rate	*9,300 req/s*				*18,478 req/s*				*25,407 req/s*				*32,582 req/s*			
Latency	μ	50%	90%	σ	μ	50%	90%	σ	μ	50%	90%	σ	μ	50%	90%	σ
Random	3.95	1.78	11.32	6.99												
R-CHash	0.79	0.53	1.46	2.67	1.01	0.57	1.98	3.58								
R-HRW	0.81	0.53	1.49	2.83	1.07	0.57	2.28	3.22								
LR-CHash	0.68	0.44	1.17	2.50	0.87	0.51	1.82	2.74	1.19	0.60	2.47	3.79				
LR-HRW	0.68	0.44	1.18	2.50	0.90	0.51	1.89	3.13	1.27	0.64	2.84	3.76				
CDR	1.16	0.52	1.47	5.96	1.35	0.55	1.75	6.63	1.86	0.63	4.49	6.62	2.37	1.12	5.19	7.21
FDR	1.10	0.52	1.48	5.49	1.35	0.54	1.64	6.70	1.87	0.62	3.49	6.78	2.22	0.87	4.88	7.12
FDR-Global	0.78	0.50	1.42	2.88	0.97	0.54	1.58	5.69	1.11	0.56	1.86	5.70	1.35	0.66	2.35	6.29

Table 5: Response Latency of Different Strategies under Normal Load. μ — Mean, σ — Standard Deviation.

about the distribution.

Figure 18 plots the cumulative distribution of latencies at four request rates: the maximums for Random, R-HRW, LR-HRW, and CDR (the algorithm in each category with the smallest maximum throughput). The *x*-axis is in log scale and shows the time needed to complete requests. The *y*-axis shows what fraction of all requests finished in that time. The data in Table 5 gives mean, median, 90th percentile and standard deviation details of response latencies at our comparison points.

The response time improvement from exploiting locality is most clearly seen in Figure 18a. At Random's capacity, most responses complete under 4 seconds, but a few responses take longer than 40 seconds. In contrast, all other strategies have median times almost one-fourth that of Random, and even their 90th percentile results are less than Random's median. These results, coupled with the disk utilization information, suggest that most requests in the Random scheme are suffering from disk delays, and that the locality improvement techniques in the other schemes are a significant benefit.

The benefit of FDR over CDR is visible in Figure 18d, where the plot for FDR lies to the left of CDR. The statistics also show a much better median response time, in addition to better mean and 90th percentile numbers. FDR-Global has better numbers in all cases than CDR and FDR, due to its perfect knowledge of server load status.

An interesting observation is that when compared to the static schemes, dynamic schemes have worse mean times but comparable/better medians and 90th percentile results. We believe this behavior stems from the time required to serve the largest files. Since these files are less popular, the dynamic schemes replicate them less than the static schemes do. As a result, these files are served from a smaller set of servers, causing them to be served more slowly than if they were replicated more widely. We do not consider this behavior to be a significant drawback, and note that some research explicitly aims to achieve this effect [11, 12].

Scalability. Robustness not only comes from resilience with certain resources, but also from good scalability with increasing resources. We repeat similar experiments with different number of servers, from 8 to 128, to test how well these strategies scale. The number of server-side routers is not changed, but instead, more servers are attached to each server router as the total number of servers increases.

We plot system capacity against the number of servers in Figure 19. They all display near-linear scalability, implying all of them are reasonably good strategies when the system becomes larger. Note, for CDR and FDR with 128 servers, our original network provision is a little small. The bottleneck in that case is the link between the server router and backbone router, which is 622Mbps. In this scenario, each server router is handling 16 servers, giving each server on average only 39Mbps of traffic. At 600 reqs/s, even an average size of 10KB requires 48Mbps. Under this bandwidth setup, CDR and FDR yield similar system capacity as LR-CHash and LR-HRW, and all these 4 strategies saturate server-router-to-backbone links. To

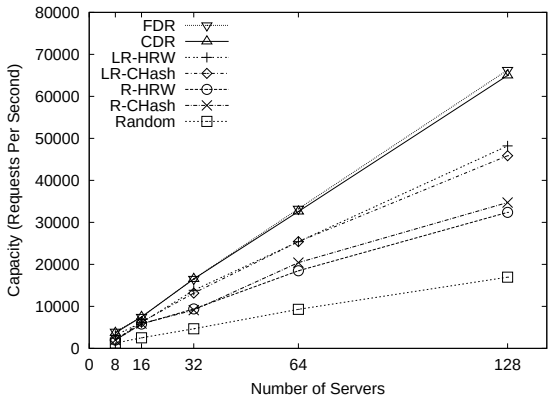


Figure 19: System Scalability under Normal Load

remedy this situation, we run simulations of 128 servers for all strategies with doubled bandwidth on both the router-to-backbone and backbone links. Performance numbers of 128 servers under these faster links are plotted in the graph instead. This problem can also be solved by placing fewer servers behind each pipe and instead spreading them across more locations.

3.4.2 Behavior Under Flash Crowds

Having established that our new algorithms perform well under normal workloads, we now evaluate how they behave when the system is under a flash crowd or DDoS attack. To simulate a flash crowd, we randomly select 25% of the 1,000 clients to be *intensive* requesters, where each of these requesters repeatedly issues requests from a small set of pre-selected URLs with an average size of about 6KB.

System Capacity. Figure 20 depicts the system capacity of 64 servers under a flash crowd with a set of 10 URLs. In general, it exhibits similar trends as the no-attack case shown in Figure 17. Importantly, the CDR and FDR schemes still yield the best throughput, making them most robust to flash crowds or attacks. Two additional points deserve more attention.

First, FDR now has a similar capacity with CDR, but still is more desirable as it provides noticeably better latency, as we will see later. FDR's benefit over R-CHash and R-HRW has grown to 91% from 60% and still outperforms LR-CHash and LR-HRW by 22%.

Second, the absolute throughput numbers tend to be larger than the no-attack case, because the workload is also different. Here, 25% of the traffic is now concentrated on 10 URLs, and these attack URLs are relatively small, with an average size of 6KB. Therefore, relative difference among different strategies within each scenario yields more useful information than simply comparing performance numbers across these two scenarios.

Response Latency. The cumulative distribution of response latencies for all seven algorithms under attack are shown in Figure 21. Also, the full statistics for all seven algorithms and FDR-Global are given in Table 6. As seen from the figure and table, R-CHash, R-HRW, LR-CHash, LR-HRW CDR and FDR still have far better latency than Random, and static schemes are a little better than CDR and FDR at Random, R-HRW's

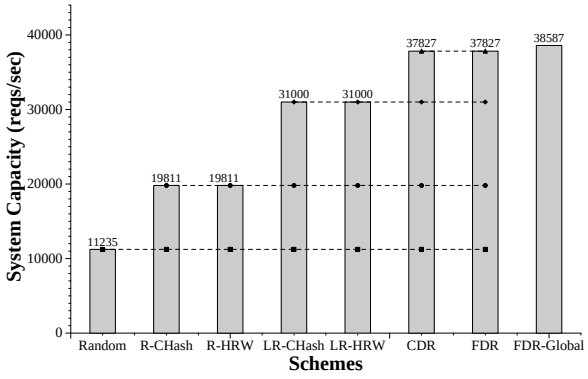


Figure 20: Capacity Comparison Under Flash Crowds

and LR-HRW's failure points; and LR-CHash and LR-HRW yields slightly better latency than R-CHash and R-HRW.

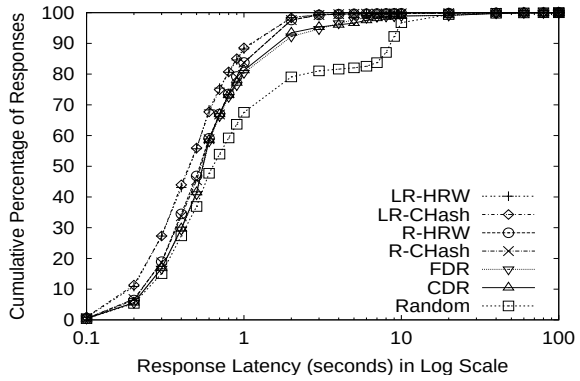
As we explained earlier, CDR and FDR adjust the server replica set in response to request volume. The number of replicas that serve attack URLs increases as the attack ramps up, which may adversely affect serving non-attack URLs. However, the differences in the mean, median, and 90-percentile are not large, and all are probably acceptable to users. The small price paid in response time for CDR and FDR brings us higher system capacity, and thus, stronger resilience to various loads.

Req Rate	*11,235 req/s*				*19,811 req/s*				*31,000 req/s*				*37,827 req/s*			
Latency	μ	50%	90%	σ	μ	50%	90%	σ	μ	50%	90%	σ	μ	50%	90%	σ
Random	2.37	0.64	8.57	5.29												
R-CHash	0.73	0.53	1.45	2.10	0.81	0.53	1.57	2.59								
R-HRW	0.73	0.52	1.45	2.11	0.76	0.52	1.51	2.51								
LR-CHash	0.62	0.45	1.15	1.70	0.67	0.45	1.23	2.42	0.96	0.52	1.86	3.55				
LR-HRW	0.63	0.45	1.18	1.80	0.67	0.46	1.26	2.65	1.07	0.53	2.19	3.52				
CDR	1.19	0.55	1.72	5.40	1.25	0.55	1.86	5.51	1.80	0.76	4.35	6.08	2.29	1.50	4.20	6.41
FDR	1.22	0.55	1.81	5.71	1.18	0.55	1.83	5.27	1.64	0.66	3.57	5.95	2.18	1.14	4.15	6.63
FDR-Global	0.91	0.55	1.66	4.09	0.90	0.53	1.60	4.59	0.98	0.54	1.74	5.08	1.20	0.56	1.99	5.53

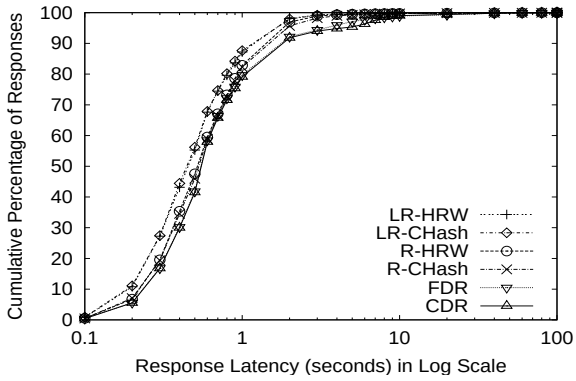
Table 6: Response Latency of Different Strategies under Flash Crowds. μ — Mean, σ — Standard Deviation.

Scalability. We also repeat the scalability test under flash crowd or attack, where 250 clients are *intensive* requesters that repeatedly request 10 URLs. As shown in Figure 22, all strategies scale linearly with the number of servers. Again in 128 server case, we use doubled bandwidth on the router-to-backbone and backbone links.

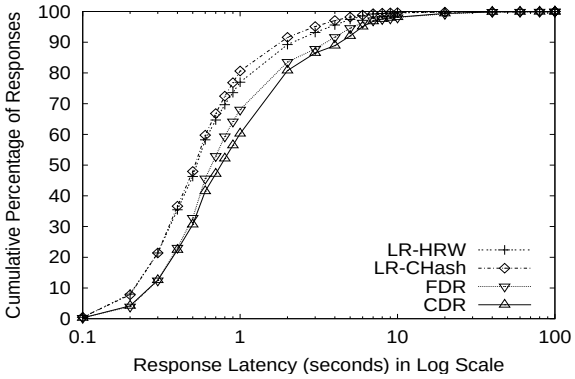
Various Flash Crowds. Throughout our simulations, we have seen that a different number of *intensive* requesters, and a different number of hot or attacked URLs, have an impact on system performance. To further investigate this issue, we carry out a series of simulations by varying both the number of intensive requesters and the number of hot URLs. Since it is impractical to exhaust all possible combinations, we choose two classes of flash crowds. One class has a single hot URL of size 1KB. This represents a small home page of a website. The other class has 10 hot URLs averaging 6KB, as before. In both cases, we vary



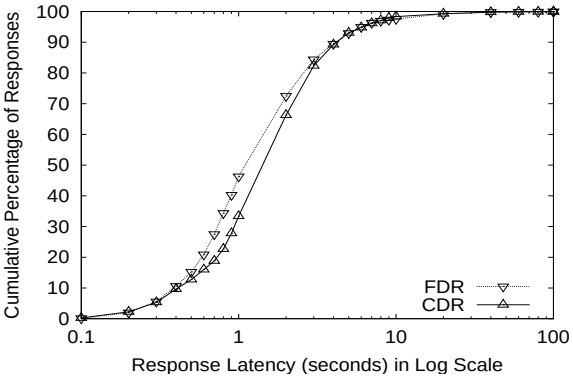
(a) Random limit: 11,235 req/s



(b) R-HRW limit: 19,811 req/s



(c) LR-HRW limit: 31,000 req/s



(d) CDR limit: 37,827 req/s

Figure 21: Response Latency Distribution under Flash Crowds

the percentage of the 1000 clients that are intensive requesters from 10% to 80%. The results of these two experiments are shown in Figures 23 and 24, respectively.

In the first experiment, as the portion of *intensive* requesters increases, more traffic is concentrated on this one URL, and the request load becomes more unbalanced. Random, CDR and FDR adapt to this change well and yield increasing throughput. This benefit comes from their ability to spread load across more servers. However, CDR and FDR behave better than Random because they not only adjust the server replica set on demand, but also maintain server locality for less popular URLs. In contrast, R-HRW, R-CHash, LR-HRW and LR-CHash suffer with more intensive requesters or attackers, since their fixed number of replicas for each URL cannot handle the high volume of requests for one URL. In the 10-URL case, the change in system capacity looks similar to the 1-URL case, except that due to more URLs being intensively requested or attacked, FDR, CDR and Random cannot sustain the same high throughput. We continue to investigate the effects of more attack URLs and other strategies.

Another possible DDoS attack scenario is to randomly select a wide range of URLs. In the case that these URLs are valid, the dynamic schemes “degenerate” into one server for each URL. This is the desirable

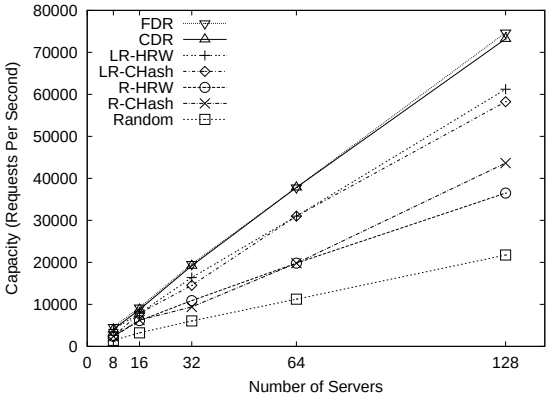


Figure 22: System Scalability under Flash Crowds

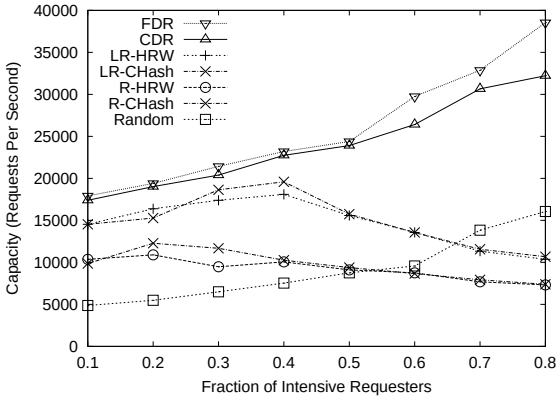


Figure 23: 1 Hot URL, 32 Servers, 1000 Clients

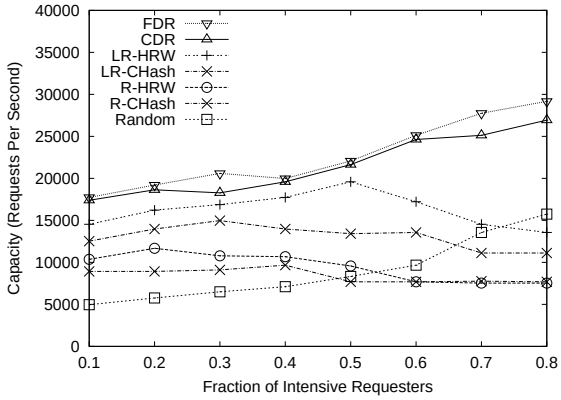


Figure 24: 10 Hot URL, 32 Servers, 1000 Clients

behavior for this attack as it increases the cache hit rates for all the servers. In the event that the URLs are invalid, and the servers are actually reverse proxies (as is typically the case in a CDN), then these invalid URLs are forwarded to the server-of-origin, effectively overloading it. Servers must address this possibility by throttling the number of URL-misses they forward.

To summarize, under flash crowds or attacks, CDR and FDR sustain very high request volumes, making overloading the whole system significantly harder and thereby greatly improving the CDN system's overall robustness.

3.4.3 Proximity

The previous experiments focus on system capacity under different loads. We now compare the strategies that factor network closeness into server selection—Static (NP-CHash), Static+Load (NPLR-CHash), and Dynamic (NP-FDR)—with their counterparts that ignore proximity. We test the 64 server case in the same scenarios as in Section 3.4.1 and 3.4.2.

Table 7 shows the capacity numbers of these strategies under both normal load and flash crowds of

Category	System Capacity (reqs/sec)		
	Scheme	Normal	Flash Crowd
Static	*NP-CHash*	14409	14409
	R-CHash	20411	19811
Static +Load	*NPLR-CHash*	24173	30090
	LR-CHash	25407	31000
Dynamic	*NP-FDR*	31000	34933
	FDR	33237	37827

Table 7: Proximity's Impact on Capacity

250 intensive requesters with 10 hot URLs. As we can see, adding network proximity into server selection slightly decreases systems capacity in the case of NPLR-CHash and NP-FDR. However, the throughput drop of NP-CHash compared with R-CHash is considerably large. Part of reason is that in LR-CHash and FDR, server load information already conveys the distance of a server. However, in the R-CHash case, the redirector randomly choosing among all replicas causes the load to be evenly distributed, while NP-CHash puts more burden on closer servers, resulting in unbalanced server load.

Req Rate	*9,300 req/s*				*14,409 req/s*				*24,173 req/s*				*31,000 req/s*			
Latency	μ	50%	90%	σ	μ	50%	90%	σ	μ	50%	90%	σ	μ	50%	90%	σ
Random	3.95	1.78	11.32	6.99												
NP-CHash	0.66	0.42	1.21	2.20	0.76	0.44	1.51	2.30								
R-CHash	0.79	0.53	1.46	2.67	0.82	0.56	1.63	2.50								
NPLR-CHash	0.57	0.36	0.93	2.00	0.68	0.39	1.33	2.34	1.34	0.55	2.63	4.73				
LR-CHash	0.68	0.44	1.17	2.50	0.71	0.48	1.43	2.19	1.04	0.50	1.95	3.44				
NP-FDR	0.70	0.50	1.42	1.63	0.67	0.49	1.33	1.56	0.80	0.49	1.55	2.82	1.08	0.53	1.96	3.54
FDR	1.10	0.52	1.48	5.49	1.25	0.54	1.71	5.87	1.60	0.57	2.10	6.84	1.88	0.59	3.72	7.25

Table 8: Proximity's Impact on Response Latency under Normal Load. μ — Mean, σ — Standard Deviation.

Req Rate	*11,235 req/s*				*14,409 req/s*				*30,090 req/s*				*34,933 req/s*			
Latency	μ	50%	90%	σ	μ	50%	90%	σ	μ	50%	90%	σ	μ	50%	90%	σ
Random	2.37	0.64	8.57	5.29												
NP-CHash	0.61	0.42	1.15	1.76	0.63	0.41	1.08	2.34								
R-CHash	0.73	0.53	1.45	2.10	0.73	0.52	1.38	2.50								
NPLR-CHash	0.53	0.36	0.90	1.75	0.55	0.35	0.91	2.29	1.29	0.61	2.65	3.94				
LR-CHash	0.62	0.45	1.15	1.70	0.64	0.44	1.13	2.56	0.90	0.49	1.73	3.44				
NP-FDR	0.70	0.50	1.45	1.68	0.66	0.45	1.34	1.63	0.81	0.47	1.64	2.55	0.99	0.51	1.92	3.26
FDR	1.22	0.55	1.81	5.71	1.07	0.54	1.67	5.47	1.60	0.66	3.49	5.90	1.84	0.78	4.15	6.31

Table 9: Proximity's Impact on Response Latency under Flash Crowds. μ — Mean, σ — Standard Deviation.

We further investigate the impact of network proximity on response latency. In Table 8 and 9, we show the latency statistics under both normal load and flash crowds. As before, we choose to show numbers at the capacity limits of Random, NP-CHash, NPLR-CHash and NP-FDR. We can see that when servers are not loaded, all schemes with network proximity taken into consideration—NP-CHash, NPLR-CHash and NP-FDR—yield better latency. When these schemes reach their limit, NP-CHash and NP-FDR still demonstrate significant latency advantage over R-CHash and FDR, respectively.

Interestingly, NPLR-CHash underperforms LR-CHash in response latency at its limit of 24,173 req/s and 30,090 req/s. NPLR-CHash is basically LR-CHash using effective load. When all the servers are not

loaded, it redirects more requests to nearby servers, thus shortening the response time. However, as the load increases, in order for a remote server to get a share of load, a local server has to be much more overloaded than the remote one, inversely proportional to their distance ratio. Unlike NP-FDR, there is no load threshold control in NPLR-CHash, so it is possible that some close servers get significantly more requests, resulting in slow processing and longer responses. In a summary, considering proximity may benefit latency, but it can also impact capacity. NP-FDR, however, achieves a good balance of both.

3.4.4 Other Factors

To determine the impact of network heterogeneity on our schemes, we explore the impact of non-uniform server network bandwidth. In our original setup, all first-mile links from the server have bandwidths of 100Mbps. We now randomly select some of the servers and reduce their link bandwidth by an order of magnitude, to 10Mbps. We want to test how different strategies respond to this heterogeneous environment. We pick representative schemes from each category: Random, R-CHash, LR-CHash and FDR and stress them under both normal load and flash crowd similar to network proximity case. Table 10 summarizes our findings on system capacities with 64 servers.

Redirection Schemes	*Portion of Slower Links*			
	Normal Load		*Flash Crowd*	
	10%	30%	10%	30%
Random	8010	8010	8449	8449
R-CHash	7471	7471	7110	7110
LR-CHash	23697	19421	26703	22547
FDR	31000	25407	34933	29496

Table 10: Capacity (reqs/sec) with Heterogeneous Server Bandwidth

From the table we can see, under both normal load and flash crowd, Random and R-CHash are hurt badly because they are load oblivious and keep assigning requests to servers with slower links thereby overload them early. In contrast, LR-CHash and FDR only suffer slight performance downgrade. However, FDR still maintains advantage over LR-CHash, due to its dynamic expanding of server set for hot URLs.

4 Network Bandwidth

The final focus of our effort was on providing different levels of service among competing network flows. Early approaches in this domain, beginning with Weighted Fair Queuing (WFQ) and culminating in the Integrated Services architecture [17, 47, 52], were able to make strong promises about the level of service provided to a given flow, but at the expense of scalability since routers must maintain per-flow state. Subsequent development of the Differentiated Services architecture [36, 8, 7, 56, 13], segregated flows into a small number of service classes (making the solution scalable), but at the expense of being able to make only relative statements about the service a given flow receives.

This section proposes an intuitively simple alternative based on probabilistic packet scheduling (PPS). It works as follows. Each router in the network defines its own currency in terms of *tickets*, and assigns these tickets to its inputs based on the contractual agreements with other routers. A flow is first tagged at a TCP source with tickets—in the appropriate local currency—that represent the relative share of bandwidth it should receive. At each hop, the flow's tickets are traded for a new set of tickets—in terms of the target

router's currency—according to an appropriate currency exchange rate. Routers probabilistically decide when to forward/drop packets based on the number of tickets and the current congestion level. When multiple flows share the same bottleneck link, PPS ensures that each flow obtains a bandwidth that is proportional to the tickets associated with that flow.

Our approach defines a new point in the design space for providing different level of service to competing flows. Like DiffServ, PPS scales well since it does not require per-flow state, and it makes only relative differentiations among flows. However, PPS differs from DiffServ in the manner in which it allocates bandwidth when there is a disparity between the link capacities and network traffic load. DiffServ installs service profiles at end hosts to represent the target rates of flows. When the links have excess bandwidth or when they fail to match the target rates, the link bandwidth is allocated in a random manner, an observation that is supported by our simulations. PPS, on the other hand, always ensures that a flow will receive its proportional share of bandwidth at its bottleneck link.

More recent work, such as Core-Stateless Fair Queueing (CSFQ) [51] and CHOKe [41], have focused on approximating fair bandwidth allocation in a stateless way. However, PPS has the advantage of being able to provide different levels of service for flows traversing multiple domains. Although weighted CSFQ allows one to assign different weights to different flows, it does not achieve proportional bandwidth allocation for TCP flows as well as PPS. Also, CSFQ assigns each flow the same weight at all routers; it cannot accommodate situations where the relative weights of flows differ from router to router. In contrast, PPS allows a flow to trade tickets it is granted in the source domain for an equitable number of tickets in each target domain along its path. (This trade is governed by an exchange rate that is easily computed by the routers connecting the two domains, based on both static inter-domain service agreements and the dynamic traffic loads.) In PPS, each domain is free to make its own decision regarding bandwidth allocation, and these decisions can be insulated from other domains. This is applicable to the implementation in today's Internet due to the autonomy of service providers. Additionally, bandwidth allocation can be controlled through either sender-based or receiver-based schemes.

The rest of this section describes PPS in more detail and presents the results of simulations that show that PPS achieves a proportional allocation of bandwidth among competing TCP flows. It concludes with a discussion of the relative strengths and weaknesses of our approach, as compared with the alternatives.

4.1 Algorithm

Our goal is to develop an algorithm that enables networks to provide different levels of service in the framework of best effort delivery. The algorithm needs to satisfy the following properties. First, it should ensure that each network flow receives a share of the bandwidth that is consistent with contractual agreements. Second, the bandwidth allocations should be reactive to dynamic changes in network traffic. Third, the algorithm should not degraded link utilization.

In this section we describe an algorithm with these properties. It consists of three components: a packet tagger, a relabeler, and a packet scheduler. Initially, each packet is tagged with some number of tickets by a TCP source. The tag is updated at each hop along the path according to some local currency exchange rate. The tag is then used to determine whether or not to drop the packet should it encounter congestion. The resulting algorithm attains proportional bandwidth allocation without requiring routers to maintain per-flow state.

The primary result of this work is that proportional bandwidth allocation for TCP flows can be achieved by strategically dropping a small number of packets. This result is surprising given the following aspects of the algorithm. First, the algorithm employs a variety of rate estimation algorithms each of which is

fundamentally inaccurate. Second, when a packet is dropped, TCP sources react drastically by halving their transmission rates; normally routers cannot communicate with the TCP sources to make small changes to their transmission rates. Third, when a flow does not require its proportional share, a scenario that is likely to occur frequently, the algorithm proportionally distributes the excess bandwidth amongst the remaining flows. This ability to reallocate bandwidth in a proportional manner requires the PPS system to dynamically identify the excess bandwidth available at the routers, an activity that could introduce further inaccuracies. In spite of the above-mentioned characteristics and the resulting complex interactions between the various mechanisms that constitute the algorithm, a well-engineered PPS system can achieve proportional bandwidth allocation with a great level of accuracy.

4.1.1 Tickets

There are two entities of interest in the network—end hosts and routers—both of which are configured with a currency in terms of tickets, and assign some number of tickets-per-second (t/s) to their inputs based on some contractual agreement. For a router, the inputs would be the various incoming links, while for an end host, the inputs would correspond to the TCP sources resident on the host. Suppose entity *P* issues OutTktRate t/s to input *A*. Packets arriving from *A* would be tagged with tickets in *P*'s currency, but these tickets should not exceed OutTktRate t/s.

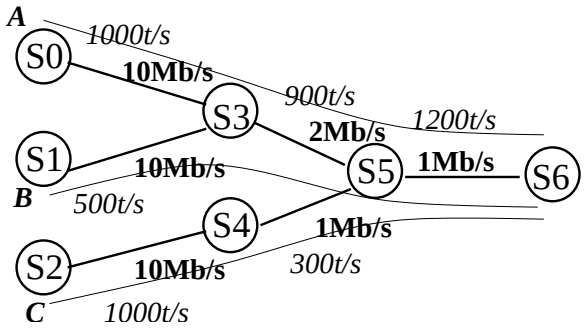


Figure 25: Example of sender-based scheme where flows A, B and C share the same bottleneck (S_5, S_6)

Figure 25 shows an example. The link capacities and ticket allocations for the configuration are indicated in the figure. For example, S_3 has issued, in its own currency, 1000 t/s to the link (S_0, S_3), which has a capacity of 10Mb/s. Suppose there are three TCP flows: *A* between S_0 and S_6, *B* between S_1 and S_6, and *C* between S_2 and S_6. In order for a packet to reach S_6 from S_0, it is first tagged with tickets by S_0 in S_3's currency. When it arrives at S_3, the tag is relabeled based on the currency exchange rate of S_3 and S_5 (which is 1500:900) before it can go to the next hop. This relabeling process continues until the packet reaches S_6. Based on the exchange rates, the ticket rate of *A*, *B* and *C* will be converted to 600, 300 and 300 t/s, respectively in S_5's currency. Because the bandwidth of link (S_5, S_6) is only 1Mb/s, it is the bottleneck for flows *A*, *B* and *C*. When all three flows are active, *A*, *B* and *C* should obtain 0.5Mb/s, 0.25Mb/s and 0.25Mb/s bandwidth respectively. When *B* is silent, *A* and *C* should divide the full bandwidth of link (S_5, S_6) by a 3:1 ratio. When *B* has a constant bit rate of 0.16Mb/s, which is less than its proportional share bandwidth of 0.25Mb/s, *B* should obtain 0.16Mb/s while *A* and *C* should obtain 0.56Mb/s and 0.28Mb/s, respectively.

4.1.2 Ticket Tagging

Suppose TCP source A is issued OutTktRate t/s by its controlling entity, and let the throughput of A be AvgRate packets/second (p/s) at this point in time. We simply tag the outgoing packet with OutTktRate / AvgRate tickets. That is, the tickets on the outgoing packet are inversely proportional to the instantaneous throughput of A. To estimate the instantaneous throughput, we use the TSW algorithm described in [7].

TSW measures the instantaneous throughput, AvgRate, by revising its rate estimate upon the arrival of each ACK and decaying the past history over a time period which is a configurable parameter, WinLength. It can therefore smooth the bursts of TCP traffic as well as be sensitive to instantaneous rate variations. The decision to employ a TSW rate estimation algorithm is primarily an engineering decision, and it can therefore be replaced by a different rate estimation algorithm without requiring changes to PPS.

4.1.3 Packet Scheduling

We first consider bandwidth allocation among one-hop TCP flows, then extend our discussion to multi-hop TCP flows in Sections 4.1.4 and 4.1.5. Suppose at some instant there are n active TCP flows—C_1, C_2, ..., C_n—contending for link L. They are issued $\mathsf{OutTktRate}_1$, $\mathsf{OutTktRate}_2$, ..., $\mathsf{OutTktRate}_n$ t/s by their controlling entity P. We use InTktRate and InPktRate to represent the t/s and p/s link L receives from all the flows.

Our scheduler needs to make the following guarantee: If a flow C_i tags AvgTkt = InTktRate / InPktRate tickets onto each of its packets, C_i should receive its proportional bandwidth share of InPktRate $\times$ $\mathsf{OutTktRate}_i$ / InTktRate p/s. Here, InPktRate is close to the capacity of L during congestion. If we can ensure that each flow tags approximately the same number of AvgTkt tickets onto its packets, we will achieve a proportional bandwidth allocation among them.

Ticket-Based RED (TRED). Our packet scheduling algorithm (Ticket-based RED) is based on the random early detection (RED) algorithm [19]. RED has three phases representing normal operation, congestion avoidance, and congestion control, respectively. During normal operation, RED does not drop any packets. During the congestion avoidance phase, each packet drop serves to notify the TCP source to reduce its sending rate. The exact drop probability is a function of the average queue length. The average queue length is calculated using a low-pass filter from the instantaneous queue lengths, which allows transient bursts in the queue. During the congestion control phase, all incoming packets are dropped.

Like RED, TRED also has three phases. It operates in the same way as RED in the normal operation and congestion control phases. During congestion avoidance, however, TRED uses a different method to calculate the drop probability. In addition to AvgQLen, TRED uses two other variables: ExpectTkt and InTkt; the former is an estimate of the number of tickets the link expects a flow to tag onto its packets, and the latter represents the number of tickets carried by an arriving packet. The TRED algorithm operates as follows.

Upon each packet arrival:

compute AvgQLen *the same as in RED*
compute InTktRate *using TSW*
compute MinTkt *as defined below*
if InTkt $<$ K $\times$ MinTkt
    compute ExpectTktRate
    and ExpectPktRate *using TSW*

if AvgQLen $\leq$ MinThresh
    enqueue the packet
if MinThresh $<$ AvgQLen $<$ MaxThresh
    calculate probability p as in RED
    ExpectTkt = ExpectTktRate
        / ExpectPktRate
    $p = p\times$ (ExpectTkt / InTkt) ^{3}
    if $p > 1$ then $p = 1$
    drop packet with probability p
if MaxThresh $\leq$ AvgQLen
    drop the packet

where variable MinTkt represents the least number of tickets seen on an arriving packet during some interval. It is important to note that although TSW is applied on a per-flow basis in [7], we apply TSW to the aggregation of all traffic arriving on a particular link.

Now we consider what happens when the link is in congestion avoidance phase. We multiply the drop probability p by (ExpectTkt / InTkt)^3. As a result, those flows that put fewer tickets on their packets are more likely to lose packets and back off than those that tag more tickets onto their packets. When a flow backs off, its instantaneous throughput slows down, and it will begin to tag more tickets onto its packets. In the end, we expect all the flows to tag approximately ExpectTkt tickets on their packets.

As we said in the beginning of Section 4.1.3, we optimally expect each flow to tag AvgTkt = InTktRate / InPktRate tickets onto its packets, but in reality, we cannot precisely calculate AvgTkt. However, the algorithm tries to keep ExpectTkt around AvgTkt. When ExpectTkt is less than AvgTkt, which means those flows are obtaining more than their proportional share of bandwidth, AvgQLen will increase accordingly. This causes the flows to slow down, and tag more tickets onto their future packets. As a result, ExpectTkt will increase. On the other hand, when ExpectTkt is greater than AvgTkt, the algorithm will cause it to decrease. In the end, the algorithm tries to keep ExpectTkt around AvgTkt, so as to proportionally allocate bandwidth among the competing flows.

Within this overall strategy of using packet drops to nudge TCP flows to obtain proportional bandwidth shares, there are two issues to be addressed.

The first issue is how to scale the drop probability based on the number of tickets carried by the packet. We have experimented with a wide variety of functions and evaluated their effectiveness with respect to the three success metrics: proportional allocation, high link utilization, and quick convergence to an equilibrium state. When the drop probability p is multiplied by (ExpectTkt / InTkt)^n for certain values of n, the system achieves the desired goals listed above. The magnitude of n has a distinct effect on system performance. For small values of n, proportional allocation is difficult to achieve as TRED fails to sufficiently penalize those TCP flows that manage to obtain a link bandwidth that is greater than their proportional share. For large values of n, the system becomes too reactive to small changes in flow throughputs resulting in unnecessary packet drops and some unpredictable effects. Our experience is that setting $n = 3$ allows the system to achieve the desired goals. At this point, we observe that TRED shares many of the congestion avoidance properties of RED

The second issue that TRED needs to address is how to estimate ExpectTkt when some of the flows are not making full use of their proportional shares, as might be the case with Constant Bit Rate (CBR) flows. We call these *rich flows* because they put more tickets onto their packets than normal flows and could potentially skew the estimated ExpectTkt value. To address this issue, we use a simple heuristic to filter out

the packets of rich flows by considering only the packets in the range of [MinTkt, $K\times$ MinTkt]. As shown in the algorithm, we calculate ExpectTkt as the average number of tickets on only those packets that fall within this range. On the one hand, K should be large enough so that most packets of normal flows will fall into this range. From the above analysis of TRED, we know that most packets of normal flows will carry approximately the same number of tickets. On the other hand, packets of a rich flow that can only make use of less than $1/K$ of its proportional bandwidth will be filtered out. Becasue packets of some rich flows are not filtered out, the skewed ExpectTkt, which we denote ExpectTkt$'$, could be K times the correct ExpectTkt, in the worse case. That will cause the skewed dropping probability, denoted p', to be K^n times the correct p. Becasue in RED,

$p = MaxP\times$
(AvgQLen - MinThresh)/(MaxThresh - MinThresh)

Suppose with the correct ExpectTkt and p, the system reaches the equilibrium state with AvgQlen. To correct for the skewed ExpectTkt$'$ and p', AvgQlen$'$ will decrease until

(AvgQLen$'$ - MinThresh) = (AvgQLen - MinThresh) / K^n

In our simulations, we choose K to be 2.

4.1.4 Tag Relabeling

The previous sections only considered TCP flows with one hop. In real networks, a flow may go through many hops before reaching the destination. Because different entities may have their own local currencies, tickets in one currency are only meaningful to the entity that issues them. Thus, when going from one entity to another, we need to relabel the tags according to some currency exchange rate. We calculate the exchange rate at each link as follows:

XRate = OutTktRate / InTktRate

As before, InTktRate corresponds to the t/s the link receives at some instant. The OutTktRate is the t/s assigned to the link by its controlling entity. For each packet, we relabel its tag as follows:

OutTkt = InTkt $\times$ XRate

In other words, the relabeling algorithm simply converts InTkt in one entity's currency to OutTkt in the currency of the next hop entity.

4.1.5 Multi-Hop Flows

A TCP flow C may utilize multiple links along its path. If the throughput of the flow does not increase when we increase the bandwidth of all links other than L, then L is the bottleneck for C. C may have several bottlenecks or no bottlenecks at all. In the latter case, the throughput of C is limited by its upper level application not by the network. Suppose flow C originates from source A, which has been assigned OutTktRate t/s by P. Also suppose that C's bottleneck is link (S, T). Based on the per-hop exchange rates

from *A* to S, we can convert OutTktRate in *P*'s currency into $\mathsf{InTktRate}_c$ t/s in S's currency. Suppose the throughput of *C* is AvgRate, the total t/s and p/s of all flows of link (S, T) are InTktRate and InPktRate. We say flow *C* obtains its proportional share of bandwidth if:

$$\mathsf{AvgRate} = \mathsf{InPktRate} \times \mathsf{InTktRate}_c \,/\, \mathsf{InTktRate}$$

Because link (S, T) is the bottleneck of *C*, it must be congested. This means that InPktRate is close to the capacity of the link. We now explain why flow *C* obtains its proportional share of bandwidth. Since link (S, T) is congested, from the TRED algorithm we know that each packet of *C* will carry approximately AvgTkt = InTktRate / InPktRate tickets when passing through link(S, T). As a result, the bandwidth that *C* obtains on link (S, T) is approximately:

$$\begin{aligned}&\mathsf{AvgRate}\\ &\quad = \mathsf{InTktRate}_c \,/\, \mathsf{AvgTkt}\\ &\quad = \mathsf{InPktRate} \times \mathsf{InTktRate}_c \,/\, \mathsf{InTktRate}\end{aligned}$$

Because link (S, T) is the bottleneck of *C*, the throughput of *C* should equal the bandwidth that *C* obtains on the link. Thus, *C* will obtain its proportional share of bandwidth.

4.1.6 Receiver-Based Algorithm

In today's Internet, it's often the case that the receiver, rather than the sender, is a more appropriate entity to determine the level of service provided to traffic flows. For example, suppose mutiple users (receivers) are downloading files from a web server (sender). We may want to allocate the bandwidth of the contended links according to the contractual agreements that the receivers have made. In the sender-based scheme, each entity defines its currency (S-currency) and assigns some t/s in S-currency to its input links or TCP sources. In the receiver-based scheme, each entity also defines its currency (R-currency) and assigns some t/s in R-currency to its output links or TCP sinks. The idea behind the receiver-based scheme is that we try to reconstruct S-currency from R-currency for each entity and compute how many t/s in S-currency an entity should issue to its input links or TCP sources. After this, we simply run the sender-based algorithm to achieve proportional bandwidth allocation among TCP flows. The receiver-based scheme assumes symmetric routing, which means the path of ACK packets is the reverse of that of data packets.

ACK Packet Tagging and Relabeling. In the sender-based scheme, only data packets are tagged by the TCP sources and then relabeled at each hop. In the receiver-based scheme, data packets are still tagged and relabeled as before. ACK packets are also tagged by the TCP sinks and then relabeled at each hop. The difference between the tags on data packets and the tags on ACK packets is that the former is used to calculate the drop probability of the data packet during congestion, while the latter is used to calculate how many t/s in S-currency an entity should assign to its input links or TCP sources.

The tagging and relabeling algorithms for ACK packets are similar to those for data packets.

Tagging algorithm at TCP sink:

- Before sending out an ACK packet, calculate the sending rate of ACK packets, AckAvgRate
- Tag the ACK packet with AckOutTktRate / AckAvgRate tickets.

Relabeling algorithm at link:

- Upon arrival of each ACK packet, calculate t/s carried by the ACK packets, AckInTktRate
- Calculate the exchange rate for an ACK packet as AckXRate = AckOutTktRate / AckInTktRate.
- Relabel the ACK packet with AckOutTkt = AckInTkt $\times$ AckXRate.

The AckOutTktRate stands for t/s that an entity assigns to its output links or TCP sinks in R-currency. We use the tickets carried by ACK packets to calculate how many t/s in S-currency that an entity should assign to its input links or TCP sources as follows:

OutTktRate = AckInTktRate

This means the ticket rate a link or TCP source can tag on its outgoing data packets equals to the ticket rate it receives from the incoming ACK packets. From the above, we can deduce that at each link:

InTktRate = AckOutTktRate

This means the ticket rate a link or a TCP sink receives from the incoming data packets equals to the ticket rate it tags on its outgoing ACK packets. So for each link, the exchange rate is:

XRate
= OutTktRate / InTktRate
= AckInTktRate / AckOutTktRate
= 1 / AckXRate

Having defined S-currency for each entity, we can now run the sender-based algorithm in the same way as described before.

Proportional Bandwidth Allocation. The bandwidth allocation for competing flows in the receiver-based algorithm is similar to that in sender-based algorithm. Suppose flow C goes from source B on end host Q to sink A on end host P. A is issued AckOutTktRate t/s by P and the bottleneck of the flow is link (S, T). Based on the per-hop AckXRate from A to T, we can convert AckOutTktRate in P's R-currency into AckInTktRate_c t/s in T's R-currency. Suppose the throughput of flow C is AvgRate p/s. Link (S, T) receives AckInTktRate t/s from the ACK packets of all the flows. The throughput of the flows is InPktRate p/s, which is close to the capacity of link (S, T) during congestion. We say C obtains its proportional share of bandwidth if it satisfies:

$\text{AvgRate} = \text{InPktRate} \times \text{AckInTktRate}_c$ / AckInTktRate

For example, in Figure 26 there are three TCP flows: A from S_6 to S_0, B from S_6 to S_1, and C from S_6 to S_2. The bandwidth of link (S_6, S_5) is 1Mb/s, which is the bottleneck of the 3 flows. When all three flows are active, A, B, and C should obtain 0.5Mb/s, 0.25Mb/s and 0.25Mb/s bandwidth, respectively.

4.1.7 Ticket Policing

As discussed in Section 4.1.2 and 4.1.4, a TCP source or link tags each outgoing packet subject to the constraint that the rate at which tickets are consumed does not exceed OutTktRate t/s. To ensure the source

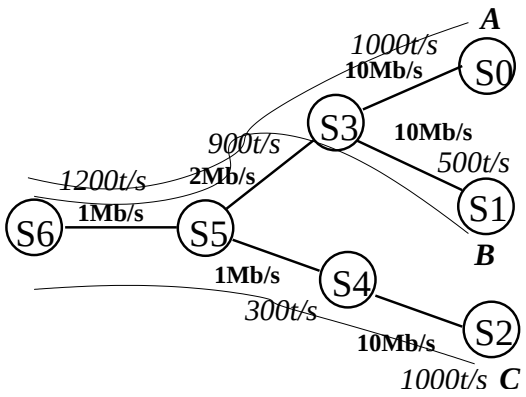


Figure 26: Example of receiver-based scheme where flows A, B and C share the same bottleneck (S_6, S_5)

or link adheres to this rate, we measure the actual ticket consuming rate, ActualTktRate, and then adjust the amount of tickets tagged to the packet as follows:

At source:

OutTkt = OutTktRate / AvgRate ×
OutTktRate / ActualTktRate

At links:

OutTkt = InTkt × XRate ×
OutTktRate / ActualTktRate

Without such an adjustment, a source or link may have a higher or lower ticket sending rate than was allocated to it.

4.2 Simulation Results

This section reports the results of several simulations designed to evaluate the algorithm's ability to proportionally allocate bandwidth among TCP flows. We use the ns-2 network simulator for our simulations [38]. We conducted each experiment on both sender-based and receiver-based scheme. The configurations of sender-based and receiver-based scheme are similar for each experiment except that the TCP source and sink are reversed, so we shall expect their simulation results to be almost the same for each experiment. In all experiments, the WinLength parameter used in the TSW algorithm is set to 5 seconds [7]. In RED, the MinThresh and MaxThresh are set to 5 and 55 packets, and the maximum dropping probability is 0.2. All experiments run for 200 seconds of simulated time.

4.2.1 One-Hop Configuration

Our first experiment studies our algorithm when there is a single congested link. We use the configuration shown in Figure 27, where 30 TCP flows share a 4.65Mb/s bottleneck link (P, Q). The flows are assigned an incremental number of t/s, ranging from 100 to 3000. The RTT for all flows is 26ms; we study the influence of RTT separately in another experiment. The throughputs are measured over the whole simulation. As

shown in Figure 28, the achieved throughput is proportional to the number of t/s given to each flow. The link utilization of (P, Q) is 99%.

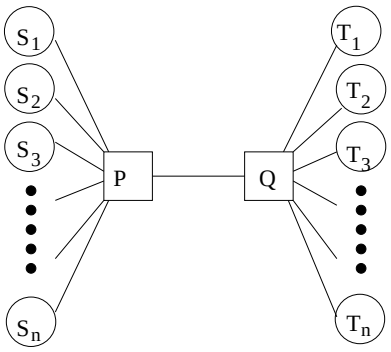


Figure 27: One-hop configuration. All *n* flows share the same bottleneck (P, Q)

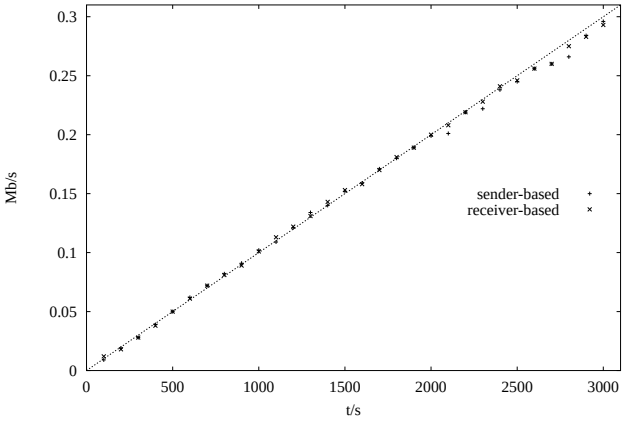


Figure 28: Bandwidth allocation for one-hop configuration. The x and y coordinate of each point represent the t/s issued to the flow and measured throughput of the flow. The achieved throughput is proportional to the number of t/s given to each flow.

4.2.2 Multi-Hop Configuration

We next study the bandwidth allocation among multi-hop flows. Figure 29 shows the network configuration we used. It has 20 flows: A_1 - A_{10} and B_1 - B_{10}. All flows share the 1.65Mb/s bandwidth of bottleneck link (P_3, P_4). Because link (P_1, P_3) is assigned twice as many t/s as link (P_2, P_3) by P_3, we expect that flow A_i obtains twice as much bandwidth as B_i ($1 \leq i \leq 10$). Also among either A_1 to A_{10} or B_1 to B_{10}, the throughput of each flow should be proportional to the number of t/s given to that flow. As shown in Figure 30 they do. The link utilization of (P_3, P_4) is 99%.

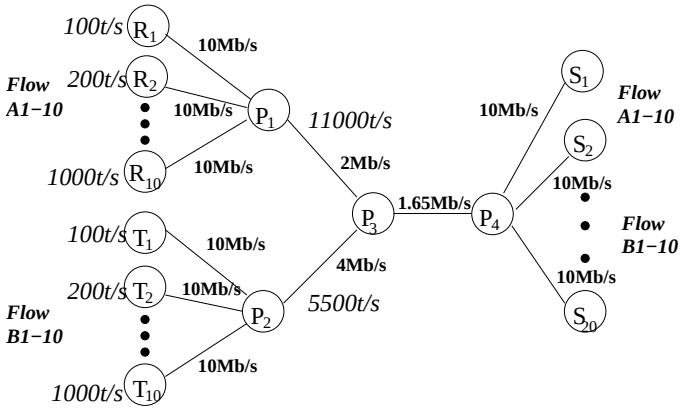


Figure 29: Multi-hop configuration. All 20 flows share the same bottleneck (P_3,P_4)

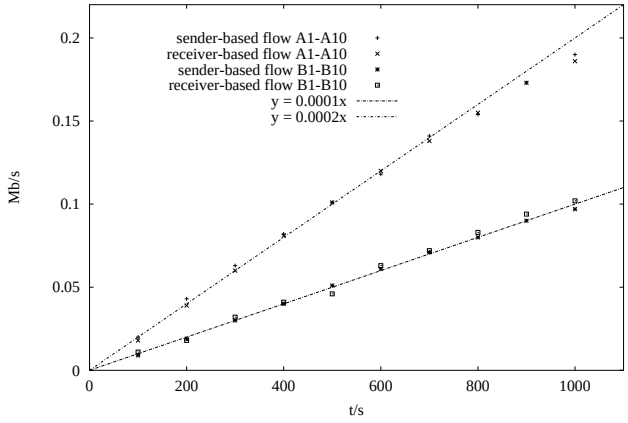


Figure 30: Bandwidth allocation for multi-hop configuration. Flow A_i obtains twice as much bandwidth as B_i ($1 \leq i \leq 10$).

4.2.3 Proportionally Sharing Unused Capacity

Sometimes a flow cannot make full use of its proportional share of bandwidth because the application generates bytes at a lower rate. The unused bandwidth should be proportionally allocated among the other flows so as to achieve high link utilization. To test the ability of our algorithm to achieve high link utilization in a proportional way, we again use the configuration from Figure 29, but this time the traffic from flow A_i, B_i ($6 \leq i \leq 10$) are generated by an application that transmits at a fixed rate of 0.03Mb/s, less than their share of bandwidth, so each of them should obtain 0.03Mb/s bandwidth. We expect flows A_i, B_i ($1 \leq i \leq 5$) to divide the excess bandwidth proportionally. As shown in Figure 31, this is exactly what happens. The link utilization of (P_3, P_4) is 99%.

4.2.4 Variable Traffic

This experiment evaluates how well the algorithm adjusts to variations in the source sending rate. We use the same configuration as in Figure 25, but when the simulation begins, only flow B and C are active; flow

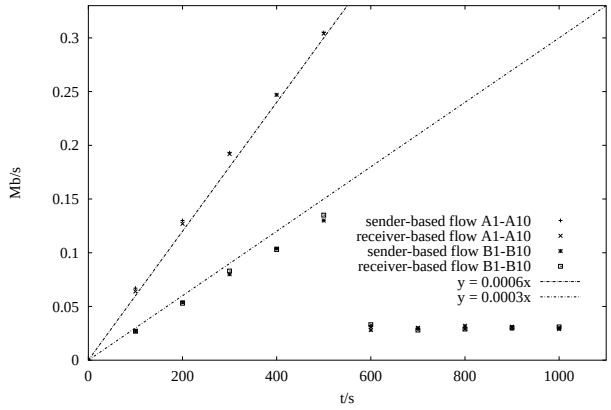


Figure 31: Proportional sharing unused bandwidth. Each of flows A_i, B_i ($6 \leq i \leq 10$) consumes 0.03Mb/s bandwidth. Flows A_i, B_i ($1 \leq i \leq 5$) divide the excess bandwidth proportionally.

A becomes active after 100 seconds. The results are shown in Figure 32. As the plot clearly shows, B and C obtain approximately 0.75Mb/s and 0.25Mb/s of bandwidth from bottleneck link (S_5, S_6) in the first 100 seconds, because they have 900 and 300 t/s in S_5's currency, respectively. After A starts up, A, B and C quickly converge to 0.5Mb/s, 0.25Mb/s and 0.25Mb/s, respectively.

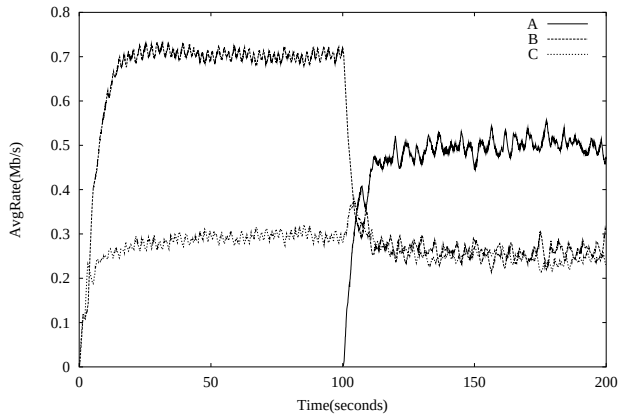


Figure 32: Adapting to new traffic. The x-axis is time and the y-axis is instantaneous throughput. Flows B and C obtain approximately 0.75Mb/s and 0.25Mb/s of bandwidth from bottleneck (S_5, S_6) in the first 100 seconds. After A starts up, A, B and C converge to 0.5Mb/s, 0.25Mb/s and 0.25Mb/s, respectively

4.2.5 Variable Ticket Allocation

Our algorithm can flexibly control bandwidth allocation among TCP flows by dynamically adjusting the rate at which tickets are issued. This permits an application to adjust its ticket share in an effort to maintain a certain transmission speed. To see this, we again use the topology in Figure 25, where in the beginning, each link is issued the same amount of tickets. After 100 seconds, the t/s issued to link (S_0, S_3) changes

from 1000 to 600 and the t/s issued to link (S_1, S_3) changes from 500 to 900. We expect the throughput of A to change from 0.5Mb/s to 0.3Mb/s, and the throughput of B to change from 0.25Mb/s to 0.45Mb/s. The throughput of C should not change. As can be seen in Figure 33, the system behaves as expected. The important point is that our algorithm keeps bandwidth allocation decisions local. That is, the variation of A and B has virtually no influence on C.

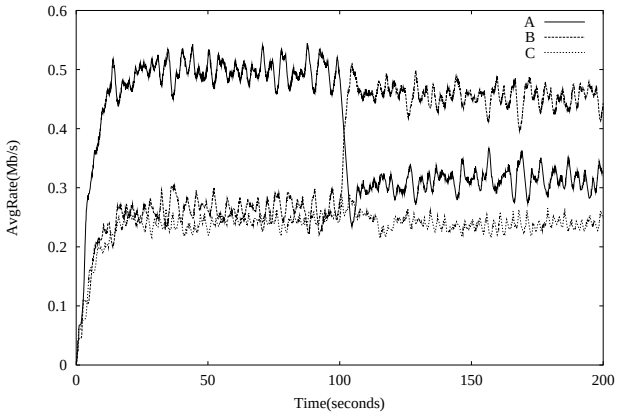


Figure 33: Bandwidth allocation as ticket rates change. As the t/s issued to link (S_0, S_3) changes from 1000 to 600 and the t/s issued to link (S_1, S_3) changes from 500 to 900, the throughput of A to change from 0.5Mb/s to 0.3Mb/s, and the throughput of B to change from 0.25Mb/s to 0.45Mb/s. The throughput of C does not change.

4.2.6 Multiple Output Links

In all the experiments up to this point, the flows share a common bottleneck link, and each router has only one output link. In this experiment, we study how the algorithm behaves when different flows have different bottlenecks and routers have multiple output links. We ran 2 experiments using the topology given in Figure 34. In this scenario, there are 4 flows—A, B, C, and D—running between (S_0, S_9), (S_1, S_8), (S_2, S_9), and (S_3, S_9), respectively.

In the first experiment, we set the bandwidth of (S_7, S_9) to 0.9Mb/s. The bottleneck of flows A, C and D is (S_7, S_9), while the bottleneck of flow B is (S_4, S_6). The 2000, 1400 and 700 t/s of flow A, C and D are converted to 2400, 800 and 400 t/s in S_7's currency. So A, C and D should divide the bandwidth of (S_7, S_9) by 6:2:1. Moreover, since A is a rich flow of (S_4, S_6) while B is not, B should obtain the remaining 0.6Mb/s bandwidth of (S_4, S_6). The measured results are shown in Table 11.

Note that although flow A has twice as many t/s as B, the actual bandwidth A obtains is almost the same as B. This may seems unfair at first glance, but because A and B are going to different destinations, they have different bottlenecks in the network. The throughput of A is limited by (S_7, S_9), so it cannot make full use of its proportional share of bandwidth at (S_4, S_6), and the unused bandwidth of A is allocated to B. From this experiment, we know that the actual bandwidth obtained by a flow is related to both its assigned ticket rate and its bottleneck.

In the second experiment, we change the bandwidth of (S_7, S_9) to 1.8Mb/s. Now the bottleneck of flow A and B is (S_4, S_6), while the bottleneck of flow C and D is still (S_7, S_9). Flow A and B should divide the 1.2Mb/s bandwidth of (S_4, S_6) by 2:1. Again, although flow A, C and D have 2400, 800 and 400 t/s in S_7's

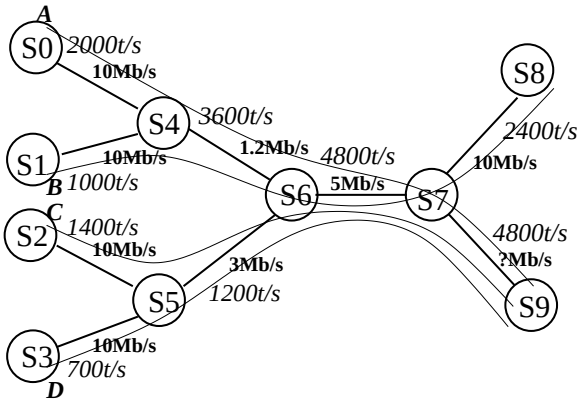


Figure 34: Complex configuration with multiple bottlenecks. When the bandwidth of (S_7, S_9) is 0.9Mb/s, A shares the same bottleneck (S_7, S_9) with C and D. When the bandwidth of (S_7, S_9) is 1.8Mb/s, A shares the same bottleneck (S_4, S_6) with B.

Flow	Sender-based Rate (Mb/s)	Receiver-based Rate (Mb/s)	Expected Rate (Mb/s)
A	0.56	0.55	0.60
B	0.64	0.65	0.60
C	0.23	0.23	0.20
D	0.11	0.12	0.10

Table 11: Multiple Bottlenecks: Scenario I. A, C and D share the same bottleneck (S_7, S_9) and proportionally divide its capacity.

currency, A is limited by its bottleneck at (S_4, S_6); C and D should share the remaining bandwidth of (S_7, S_9) by 2:1. The actual results are shown in Table 12.

4.2.7 RTT Biases

It is well-known that TCP has a bias against flows with large round trip times (RTT). To understand the relationship between our algorithm and RTT, we conduct two sets of experiments with configurations depicted in Figure 27, where 10 flows share a bottleneck link (P, Q) of 1.1Mb/s.

In scenario I, all flows are assigned 200 t/s. We first set all the RTTs to 30ms. This setting serves to demonstrate that our algorithm is able to fairly split bandwidth among competing flows. We then let the RTT of the flows vary, incrementally, from 30ms to 300ms. Optimally, each flow should obtain 0.110Mb/s bandwidth. As can be seen from Table 13, when all the RTTs are the same, our algorithm can more fairly allocate bandwidth than Reno TCP with RED. When RTTs vary, although there is still a bias against long-RTT flows in our algorithm, it's much better than that of Reno TCP. This is because when a flow with large RTT backs off, it begins to tag more tickets onto each of its packets. When contending with other flows, its packets are more likely to get through and the flow recovers to its share of bandwidth faster than Reno TCP.

In scenario II, each flow is assigned an incremental number of t/s, ranging from 100 to 1000. We first set the RTT of flow 1, which has 100 t/s, to 260ms, and the RTT of all other flows to 26ms. We then reset the experiment so that the RTT of flow 10, which has 1000 t/s, is 260ms and the RTT of all other flows is

Flow	Sender-based Rate (Mb/s)	Receiver-based Rate (Mb/s)	Expected Rate (Mb/s)
A	0.74	0.74	0.80
B	0.45	0.45	0.40
C	0.69	0.68	0.67
D	0.36	0.37	0.33

Table 12: Multiple Bottlenecks: Scenario II. A and B share the same bottleneck (S_4, S_6) and proportionally divide its capacity.

RTT ms	PPS Mb/s	Reno Mb/s	RTT ms	PPS Mb/s	Reno Mb/s
30	0.107	0.108	300	0.087	0.060
30	0.110	0.099	270	0.090	0.055
30	0.110	0.105	240	0.098	0.071
30	0.109	0.118	210	0.100	0.086
30	0.107	0.112	180	0.105	0.079
30	0.107	0.113	150	0.110	0.114
30	0.108	0.118	120	0.118	0.115
30	0.114	0.099	90	0.122	0.127
30	0.109	0.116	60	0.123	0.159
30	0.110	0.108	30	0.129	0.233
σ	0.002	0.007	σ	0.014	0.051

Table 13: Variable RTT: Scenario I. σ is the standard deviation. PPS can more fairly allocation bandwidth than Reno TCP.

26ms. As we can see from the results shown in Figure 35, a larger RTT has a more negative influence on flow 10 than flow 1. This is because the proportional share of bandwidth of flow 10 is greater than that of flow 1. When both flows back off, flow 10 loses more bandwidth than flow 1. So under a large RTT, it takes longer for flow 10 to recover to its share of bandwidth than for flow 1.

4.2.8 Comparison with DiffServ

DiffServ installs service profiles at end hosts and tags each packet with one bit (in/out) to indicate if the packet is beyond the limits set by its service profile [7]. When congestion happens, routers preferentially drop packets sent outside the profile. DiffServ works well when the link capacity matches the service profiles, but this condition is inherently hard to achieve. Because the service profiles are just expected sending rates, they do not take into account the full path taken by flows. It is possible that many flows are contending for some link in the middle of the network, or those links that were expect to be shared are temporarily idle. It is impossible to guarantee that the link capacity matches the total target profile rate of contending flows at any time at any place in the network. But in reality, the service profiles are usually associated with the user payments. A user who pays twice more than another user would expect to receive twice as much as bandwidth. So it would be reasonable to allocate bandwidth in proportion to the service profiles.

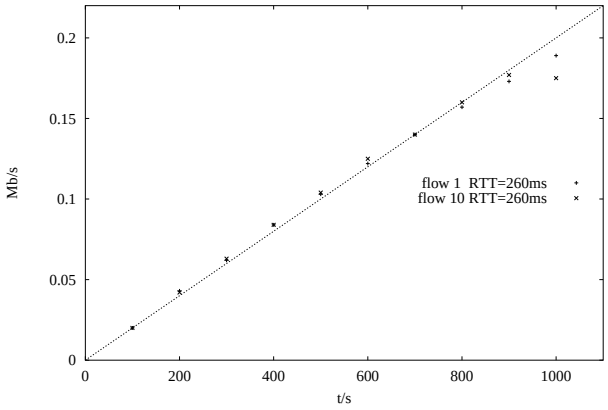


Figure 35: Variable RTT: Scenario II. A larger RTT has more negative influence on a flow with larger ticket share.

To evaluate the impact of this effect, we run a series of experiments that measure the behavior of DiffServ when there is a mismatch between link capacity and service profiles. We use the topology in Figure 27, which has 3 flows (A, B and C) contending for bottleneck link (P, Q) with a bandwidth of 1.2Mb/s. In each experiment, the ticket assignment used by PPS are at the same ratio (3:2:1) as the service profiles, and our algorithm allocates bandwidth in proportion to the service profiles of the three flows, independent of the available capacity.

Flow	Service Profile	Measured DiffServ	Service Profile	Measured DiffServ
A	0.15	0.43	1.20	0.42
B	0.10	0.40	0.80	0.43
C	0.05	0.37	0.40	0.35
Total	0.30	1.20	2.40	1.20

Table 14: DiffServ can't proportionally allocate bandwidth when profiles don't match capacity (Mb/s).

As we had expected, when the target sending rate (service profile) of A, B and C match the 1.2Mb/s capacity of the shared link, DiffServ and our algorithm work equally well. When the target rate of A, B and C are below the available link capacity, DiffServ allocates the excess bandwidth arbitrarily. When the expected sending rate of A, B and C exceed the capacity available on the link, many in-profile packets are dropped, causing DiffServ to degenerate into best effort. This is shown in Table 14.

4.2.9 Comparison with CSFQ

Stoica, Zhang, and Shenker propose Core-Stateless Fair Queueing (CSFQ)[51] to achieve fair queueing without using per-flow state in the core of an island of routers. Packets are marked with their sending rate at the edge routers, then core routers preferentially drop packets from a flow based on its fair share and the rate encoded in the packets. They also extend CSFQ to weighted CSFQ by assigning different weights to different flows.

Although CSFQ is able to fairly allocate bandwidth for TCP flows, weighted CSFQ doesn't work as well in proportionally allocating bandwidth among TCP flows. Because TCP is self-adaptive, we would expect a set of competing TCP flows to obtain approximately equal share of bandwidth under some scheduling strategy, such as CSFQ, as long as this strategy doesn't have a bias against any particular TCP flow. But that doesn't mean this scheduling strategy can also achieve proportional bandwidth allocation for TCP flows. As an example, we run weighted CSFQ[14] using the same configuration as in Section 4.2.1, except that each flow is assigned an incremental weight, ranging from 1 to 30, instead of tickets. Comparing Figure 36 with Figure 28, we can see that weighted CSFQ doesn't achieve proportional bandwidth allocation for TCP flows as well as PPS.

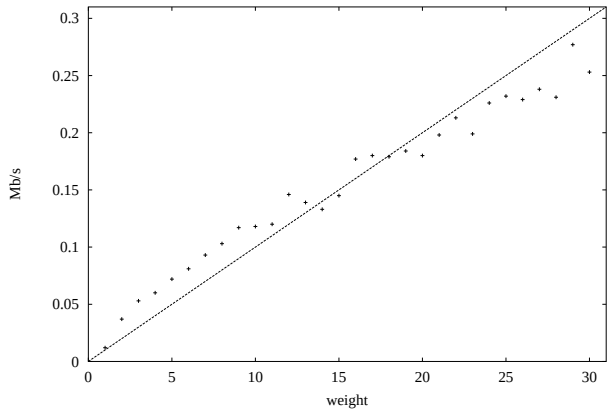


Figure 36: Bandwidth allocation for weighted CSFQ. The x and y coordinate of each point represent the weight of the flow and measured throughput of the flow.

The reason is weighted CSFQ tries to drop the portion of packets of a flow that exceed its weighted share of bandwidth. While weighed CSFQ can achieve proportional bandwidth allocation for non-responsive flows, such as UDP, with a high accuracy, it can not do equally well for TCP flows. A TCP flow will cut down its sending rate by half in response to a single packet drop [27], so dropping all those packets that exceed a flow's weighted share of bandwidth is a little too drastic for TCP and may lead to some unpredictable effects. In comparison, PPS nudges TCP in a more gentle and controlled fashion.

Weighted CSFQ can assign different weights to different flows, but each flow has the same weight at all routers. This can be viewed as a special case of PPS—all the entities have the same currency and keep all the exchange rates as 1. However, there need not be a single currency in PPS. Each entity is free to adopt its own currency, so each router has more flexibility in bandwidth allocation. For example, in Figure 25, if S_5 wishes to allocate more bandwidth to flows coming from link (S_3, S_5), it can simply assign more tickets to that link. In CSFQ, however, S_5 has to trace to S_0 and S_1 and ask them to assign more weights to their incoming flows. This would be difficult when the number of flows is large or when a flow goes through many links to reach S_5. Also in PPS, each entity is free to make its own decision regarding bandwidth allocation, and those decisions can be insulated from other entities. For example, in Figure 25, while S_3 can freely decide how many tickets should be assigned to link (S_0, S_3) and (S_1, S_3), it can not influence the allocations by S_4 and S_5. As shown in Section 4.2.5, when S_3 changes the ticket allocation, the variation of flow A and B has no influence on C, although C shares link (S_5, S_6) with A and B. But in CSFQ, if the weight of a flow is changed, it may influence any flow that share a link with it. Finally, in PPS the bandwidth allocation can be controlled through both sender-based and receiver-based schemes. This would be important in many

scenarios as we explained in the beginning of Section 4.1.6.

4.2.10 Ticket Bits

We use 8-bit tags in all the experiments reported in this section, which means the number of tickets on each packet is in the range [0, 255]. We have experimented with both fewer and more bits, and as one would expect, the more bits we use, the finer differentiation we can make among TCP flows. Also with more bits, we have more flexibility in configuring the entities. Because tickets are relabeled at each hop according to some currency exchange rate, we need to carefully configure the currency of each entity and its contractual agreements with other enties. Otherwise when trading tickets from one currency to another, it's possible that the tickets go beyond the maximum limit or drop down to 0. There are two points to make, however. First, the number of bits needed is not related to the number of hops across the network but to the exchange rates of each hop. Hence, we are not concerned that more complex topologies will require more bits as long as the exchange rates are configured in a reasonable way. Second, the number of bits needed is dependent on the number of levels of service one wants to provide at a given router. It is independent of the number of flows one is trying push through the router, because most competing flows in a router will tag approximately the same number of tickets on their packets.

From a practical point of view, Stoica and Zhang describe how the 13-bit ip_off field in IP header can be added to the 4 bits from the type of service (TOS) to create a 17-bit tag [51]. Our simulations suggest that this is enough for our approach.

References

[1] Akamai. Akamai content delivery network. http://www.akamai.com.

[2] Alteon WebSystems, Inc., San Jose, California. *ACEnic Server-Optimized 10/100/1000 Mbps Ethernet Adapters Datasheet*, August 1999.

[3] F. Baker. Requirements for IP Version 4 Routers; RFC 1812. *Internet Request for Comments*, June 1995.

[4] A. Barbir, B. Cain, F. Douglis, M. Green, M. Hofmann, R. Nair, D. Potter, and O. Spatscheck. Known CN Request-Routing Mechanisms, Feb. 2002. Work in Progress, draft-ietf-cdi-known-request-routing-00.txt.

[5] J. C. R. Bennett and H. Zhang. Hierarchical Packet Fair Queueing Algorithms. In *Proceedings of the ACM SIGCOMM '96 Conference*, pages 143–156, August 1996.

[6] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies an d Systems (USITS)*, Monterey, CA, Dec. 1997.

[7] D. Clark and W. Fang. Explicit allocation of best-effort packet delivery service. *IEEE/ACM Transactions on Networking*, 6(4):362–373, Aug. 1998.

[8] D. Clark and J. Wroclawski. An approach to service allocation in the internet. *Internet Draft*, July 1997.

[9] J. Cohen, N. Phadnis, V. Valloppillil, and K. W. Ross. Cache array routing protocol v1.1. http://ds1.internic.net/internet-drafts/draft-vinod-carp-v1-01.txt, September 1997.

[10] Compaq Computer Corporation, Intel Corporation, Microsoft Corporation. *Virtual Interface Architecture Specification, Version 1.0*, December 1997.

[11] M. Crovella, R. Frangioso, and M. Harchol-Balter. Connection scheduling in web servers. In *USENIX Symposium on Internet Technologies and Systems*, 1999.

[12] M. Crovella, M. Harchol-Balter, and C. D. Murta. Task assignment in a distributed system: Improving performance by unbalancing load (extended abstract). In *Measurement and Modeling of Computer Systems*, pages 268–269, 1998.

[13] J. Crowcroft and P. Oechslin. Differentiated end-to-end internet services using a weighted proportional fair sharing tcp. *Computer Communication Review*, 28(3), July 1998.

[14] csfq (online). http://www.cs.berkeley.edu/~istoica/csfq/.

[15] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner. Router Plugins: A Software Architecture for Next Generation Routers. In *Proceedings of the ACM SIGCOMM '98 Conference*, pages 229–240, September 1998.

[16] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small Forwarding Tables for Fast Routing Lookups. In *Proceedings of the ACM SIGCOMM '97 Conference*, pages 3–14, October 1998.

[17] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *Proceedings of ACM SIGCOMM*, pages 362–373, Aug. 1989.

[18] Digital Island. http://www.digitalisland.com.

[19] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, July 1993.

[20] L. Garber. Technology news: Denial-of-service attacks rip the Internet. *Computer*, 33(4):12–17, Apr. 2000.

[21] A. N. Habermann. *Introduction to Operating System Design*, pages 72–75. Science Research Associates, Inc., 1976.

[22] G. Hunt, E. Nahum, and J. Tracey. Enabling content-based load distribution for scalable services. Technical report, IBM T.J. Watson Research Center, May 1997.

[23] InfiniBand^{SM} Trade Association. *InfiniBand^{TM} Architecture Specification, Release 1.0*, October 2000.

[24] Intel Corporation. *IXP1200 Network Processor Datasheet*, September 2000.

[25] Intel Corporation. *IXP12EB Intel IXP1200 Network Processor Ethernet Evaluation Kit Product Brief*, 2000.

[26] Intelligent I/O (I_2O) Special Interest Group. *Intelligent I/O (I_2O) Architecture Specification, Version 2.0*, March 1999.

[27] V. Jacobson and M. Karels. Congestion avoidance and control. *Proceedings of ACM SIGCOMM*, pages 314–329, Aug. 1988.

[28] K. L. Johnson, J. F. Carr, M. S. Day, and M. F. Kaashoek. The measured performance of content distribution networks. In *Proceedings of The 5th International Web Caching and Content Delivery Workshop*, Lisbon, Portugal, May 2000.

[29] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi. Web caching with consistent hashing. In *Proceedings of the Eighth International World-Wide Web Conference*, 1999.

[30] D. R. Karger, E. Lehman, F. T. Leighton, R. Panigrahy, M. S. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing*, pages 654–663, 1997.

[31] R. Keller, S. Choi, D. Decasper, M. Dasen, G. Fankhauser, and B. Plattner. An Active Router Architecture for Multicast Video Distribution. In *Proceedings of the IEEE INFOCOM 2000 Conference*, pages 1137–1146, March 2000.

[32] T. V. Lakshman and D. Stiliadis. High Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching. In *Proceedings of the ACM SIGCOMM '98 Conference*, pages 203–214, September 1998.

[33] Mirror Image. http://www.mirror-image.com.

[34] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 217–231, December 1999.

[35] D. Mosberger and L. L. Peterson. Making Paths Explicit in the Scout Operating System. In *Proceedings of the Second USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 153–167, October 1996.

[36] K. Nichols, V. Jacobson, and L. Zhang. An approach to service allocation in the internet. *Internet Draft*, Nov. 1997.

[37] NS. (Network Simulator). http://www.isi.edu/nsnam/ns/.

[38] ns2 (online). http://www.isi.edu/nsnam/ns.

[39] V. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *USENIX Annual Technical Conference*, June 1999.

[40] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. M. Nahum. Locality-aware request distribution in cluster-based network servers. In *Architectural Support for Programming Languages and Operating Systems*, pages 205–216, 1998.

[41] R. Pan, B. Prabhakar, and K. Psounis. Choke, a stateless active queue management scheme for approximating fair bandwidth allocation. *IEEE INFOCOM*, Mar. 2000.

[42] PCI Special Interest Group, Hillsboro, Oregon. *PCI Local Bus Specification, Revision 2.2*, December 1998.

[43] L. Peterson, Y. Gottlieb, M. Hibler, P. Tullmann, J. Lepreau, S. Schwab, H. Dandelkar, A. Purtell, and J. Hartman. An OS Interface for Active Routers. *IEEE Journal on Selected Areas in Communications*, 19(3):473–487, March 2001.

[44] L. L. Peterson, S. C. Karlin, and K. Li. OS Support for General-Purpose Routers. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS–VII)*, pages 38–43, March 1999.

[45] X. Qie, A. Bavier, L. Peterson, and S. Karlin. Scheduling Computations on a Programmable Router. In *Proceedings of the ACM SIGMETRICS 2001 Conference*, pages 13–24, June 2001.

[46] RAMiX Incorporated, Ventura, California. *PMC/CompactPCI Ethernet Controllers Product Family Data Sheet*, 1999.

[47] S. Shenker, R. Braden, and D. Clark. Integrated services in the internet architecture: an overview. *Internet RFC 1633*, June 1994.

[48] T. Spalink, S. Karlin, and L. Peterson. Evaluating Network Processors in IP Forwarding. Technical Report TR–626–00, Department of Computer Science, Princeton University, November 2000.

[49] T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb. Building a Robust Network-Processor-Based Router. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, October 2001. to appear.

[50] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and Scalable Level Four Switching. In *Proceedings of the ACM SIGCOMM '98 Conference*, pages 191–202, September 1998.

[51] I. Stoica, S. Shenker, and H. Zhang. Core-stateless fair queueing: achieving approximately fair bandwidth allocations in high speed networks. *Proceedings of SIGCOMM*, pages 118–130, Aug. 1998.

[52] I. Stoica and H. Zhang. Providing guaranteed services without per flow management. *Proceedings of SIGCOMM*, pages 81–94, Aug. 1999.

[53] D. G. Thaler and C. V. Ravishankar. Using name-based mappings to increase hit rates. *IEEE/ACM Transactions on Networking*, 6(1):1–14, Feb. 1998.

[54] Vitesse Semiconductor Corporation, Longmont, Colorado. *IQ2000 Network Processor Product Brief*, 2000.

[55] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable High Speed IP Routing Lookups. In *Proceedings of the ACM SIGCOMM '97 Conference*, pages 25–36, October 1997.

[56] Z. Wang. User-share differentiation (usd) scalable bandwidth allocation for differentiated services. *Internet Draft*, Nov. 1997.

[57] D. Wetherall. Active network vision and reality: lessons from a capsule-based system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 64–79, December 1999.

Appendix A

At the end of the contract period we took on an additional project to help transition our results - and the results of others in the FTN program - to future deployment platforms. This involved implementing software infrastructure that makes it possible to test and deploy new network architectures, services, and protocols on PlanetLab, a global experimental network.

Specifically, we developed a prototype toolkit for deploying and running a packet routing infrastructure in a slice of PlanetLab. The toolkit supports both a packet forwarding data plane (e.g., the Click extensible IP forwarding engine) and one or more control protocols (e.g., the XORP Internet routing protocol suite). The toolkit allows unmodified applications running on desktop (client) machines to splice into a routing overlay, and communicate with unmodified server machines running anywhere in the Internet.

A PlanetLab slice is an overlay network of virtual machines (VM). PlanetLab currently spans 440 machines world-wide. When a slice is configured with Click and XORP, the toolkit allows DARPA performers (and other researchers) to modify and experiment with a virtual instantiation of today's Internet protocols. When a slice is deployed with custom forwarding and routing protocols, the toolkit supports experimentation with alternative routing architectures.

This effort involved four major tasks:

1. Core PlanetLab architecture to support for tunneling. This task involved adding support to the kernel for GRE (Generic Routing Encapsulation) tunnels, providing mechanisms for multiple slices to reserve key resources to implement tunnels (e.g., link bandwidth and unique tunnel identifiers), and enabling operations to create, destroy, and control tunnels.

2. Overlay ingress/egress machinery. At the overlay ingress node, this machinery interacts with client desktops via the Point-to-Point Tunneling Protocol (PPTP). At the overlay egress node, the mechanism serves as a Network Address Translator (NAT) with respect to the rest of the Internet.

3. Support for splicing desktop (client) machines into the routing overlay. This involved providing a reference implementation of PPTP, which allows unmodified application programs to slice into the overlay network, thereby using it (rather than the default Internet) to communicate with unmodified server machines.

4. Documentation and support. We have written documentation describing both the toolkit and PlanetLab in general. It also includes on-going support for developers as they use PlanetLab to demonstrate and evaluate their control plane solutions.